

Loosely-coupled fail-operational execution on embedded heterogeneous multi-cores

Rico Amslinger

Angaben zur Veröffentlichung / Publication details:

Amslinger, Rico. 2021. "Loosely-coupled fail-operational execution on embedded heterogeneous multi-cores." Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



LOOSELY-COUPLED FAIL-OPERATIONAL EXECUTION ON EMBEDDED HETEROGENEOUS MULTI-CORES

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Angewandte Informatik
der Universität Augsburg

UNIA
Universität
Augsburg
University



eingereicht von
Rico Amslinger, M.Sc.

LOOSELY-COUPLED FAIL-OPERATIONAL EXECUTION ON EMBEDDED HETEROGENEOUS MULTI-CORES

Rico Amslinger, M.Sc.

Erstgutachter: Prof. Dr. Sebastian Altmeyer

Zweitgutachter: Prof. Dr. Theo Ungerer

Tag der mündlichen Prüfung: 13. August 2021

Kurzfassung

Moderne sicherheitskritische Anwendungsbereiche für eingebettete Systeme wie autonomes Fahren benötigen Fehlertoleranz, da ein Ausfall Menschenleben gefährden kann. Gleichzeitig wird eine hohe Rechenleistung bei geringem Stromverbrauch gefordert. Ein übliches Verfahren, um dies zu erreichen, ist der Einsatz von heterogenen Multicores. Gängige Fehlertoleranzmechanismen profitieren jedoch nicht von der Heterogenität und leiden unter weiteren Nachteilen: Einige (z. B. Dual Modular Lockstep) benötigen zusätzliches Checkpointing, um Fehler zu beheben. Andere (z. B. Triple Modular Redundancy) haben einen hohen Ressourcenbedarf, was zu hohen Hardwarekosten und einem hohen Stromverbrauch führt. Weitere Ansätze (z. B. softwarebasierte Redundanz auf Prozessebene) können nicht mit dem Indeterminismus umgehen, der durch parallele Ausführung entsteht.

Zur Lösung dieser Probleme stellt diese Dissertation einen neuartigen Ansatz zur fehlertoleranten Ausführung mit Hardware-Transaktionsspeicher vor. Jeder Thread wird automatisch in Transaktionen aufgeteilt, die redundant auf zwei Kernen ausgeführt werden. Diese Transaktionen können aufgrund ihrer losen Kopplung zu unterschiedlichen Zeiten abgeschlossen werden. Die nachlaufenden Kerne werden durch die Weiterleitung von Informationen der vorauslaufenden Kerne beschleunigt, wodurch sich der Ansatz gut für heterogene Systeme eignet. Die bereits vorhandene Rückrollfähigkeit des Transaktionsspeichers wird zur Fehlerbeseitigung benutzt, wodurch der Overhead an Chipfläche und Leistung verringert wird.

Der Transaktionsspeicher wird erweitert, um für jedes Speicherwort mehrere Versionen bereitzustellen, welche verwendet werden, um identische Ergebnisse bei unterschiedlicher Ausführungsreihenfolge auf vorauslaufenden und nachfolgenden Kernen zu garantieren. Diese Versionen erlauben auch das gleichzeitige Zurücksetzen mehrerer Kerne in einen konsistenten Zustand, wenn ein Fehler auftritt. Eine Nutzung des Transaktionsspeichers zur Synchronisation ist weiterhin möglich, was die Ausführung von parallelen Anwendungen mit gemeinsam genutztem Speicher ermöglicht.

Der Ansatz für sequenzielle Anwendungen wurde in einem Simulator und der Ansatz für parallele Anwendungen auf einem FPGA evaluiert. Die Auswertung im Simulator zeigt, dass der Ansatz schneller als eine Lockstep-Konfiguration aus energieeffizienten Kernen läuft und dabei weniger Energie verbraucht als eine Lockstep-Konfiguration aus schnellen Kernen. Die FPGA-Implementierung weist eine Fehlererkennungslatenz auf, die gering genug für die meisten eingebetteten Systeme ist, und die Fehlerinjektionsanalyse zeigt, dass alle Fehler erfolgreich korrigiert werden können.

Abstract

Modern safety-critical embedded applications like autonomous driving need to be fail-operational, since failure can endanger human lives. At the same time, high performance and low power consumption are demanded. A common way to achieve this is the use of heterogeneous multi-cores. However, prevalent fault tolerance mechanisms do not benefit from the heterogeneity and suffer from further disadvantages: Some (e.g. dual modular lockstep) require supplementary checkpointing mechanisms to recover from errors. Others (e.g. triple modular redundancy) require a substantial amount of duplication, resulting in high hardware costs and high power consumption. Further approaches (e.g. software-based process-level redundancy) cannot handle the indeterminism introduced by multi-threaded execution.

To overcome these issues, this thesis presents a novel approach to fault tolerance utilizing hardware transactional memory. Each thread is automatically split into transactions, which execute redundantly on two cores. These transactions can complete at different times due to loose coupling. The trailing cores are accelerated by forwarding information from the leading cores, which makes the approach well suited for heterogeneous systems. Recovery utilizes the preexisting rollback capability of the transactional memory, which reduces the overhead in terms of chip area and performance.

The transactional memory is extended to support multiple versions of each memory word, which are used to guarantee identical outcomes in the presence of different execution schedules on leading and trailing cores. These versions also enable the simultaneous rollback of multiple cores to a consistent state if an error occurs. Use of the transactional memory for synchronization is still possible, which enables the execution of shared memory multi-threaded applications.

The single-threaded variant was evaluated in a simulator and the multi-threaded variant was evaluated on an FPGA. The single-threaded evaluation demonstrates that the approach runs faster than a lockstep configuration of energy-efficient cores, while consuming less energy than a lockstep configuration of fast cores. The multi-threaded approach exhibits an error detection latency that is low enough for most embedded systems and its fault injection analysis shows that it can successfully correct all errors.

Danksagung

Im Folgenden möchte ich mich bei einigen Personen bedanken, deren Unterstützung für die Anfertigung dieser Doktorarbeit unerlässlich war. In der Anfangsphase hat Prof. Theo Ungerer die Dissertation betreut. Nach seiner Pensionierung hat Prof. Sebastian Altmeyer die Betreuung übernommen. Ich bin sehr dankbar für die ausgezeichnete Betreuung in allen Phasen der Dissertation. Auch möchte ich Prof. Rudi Knorr dafür danken, dass er sich als Drittprüfer für die mündliche Prüfung bereitgestellt hat.

Außerdem danke ich allen Mitarbeitern des Lehrstuhls. Ich werde unsere Unterhaltungen beim Mittagessen vermissen. Mein besonderer Dank gilt meinen Kollegen Christian Piatka, Florian Haas und Sebastian Weis, die sich ebenfalls mit Transaktionspeicher beschäftigt haben. Die Kritikpunkte und Anregungen, welche in unseren häufigen Diskussionen aufgekommen sind, waren für den Fortschritt dieser Arbeit unerlässlich.

Bei meiner Familie bedanke ich mich für die fortlaufenden Unterstützung, welche diese Arbeit erst ermöglicht hat.

Rico Amslinger
Augsburg im August 2021

Table of Contents

Kurzfassung	iii
Abstract	v
Danksagung	vii
List of Abbreviations	xv
1. Introduction	1
1.1. State of the Art	2
1.2. Tight and Loose Coupling	3
1.3. Multi-Threaded Execution	5
1.4. Our Solution	6
1.5. Outline	7
1.6. Funding and Publications	8
2. Background	9
2.1. Fault Tolerance	10
2.1.1. Faults and Errors	10
2.1.2. Persistence of Faults	10
2.1.3. Impact of Errors	11
2.1.4. Measurement of Reliability	11
2.1.5. Kinds of Redundancy	11
2.1.6. Error Correction Codes	12
2.1.7. Sphere of Replication	12
2.2. Transactional Memory	13
2.2.1. Programming Model	14
2.2.2. Conflict Detection	14
2.3. FPGA and MicroBlaze	15
2.3.1. FPGA	15
2.3.2. MicroBlaze	18
2.3.3. Connectivity	19
2.4. Summary	20
3. Execution Model	21
3.1. Hardware Architecture	21

3.2. Redundant Execution	22
3.2.1. Goal of Our Approach	22
3.2.2. Scope of Our Approach	23
3.2.3. Automatic Transactions	23
3.2.4. Concept	24
3.3. Summary	25
4. Single-Threaded Fault Tolerance	27
4.1. Concept	28
4.2. Enhancement of the Transactional Memory	28
4.2.1. Automatic Transaction Bounds	28
4.2.2. Error Detection	30
4.2.3. Double Checkpoints	30
4.2.4. Conflict Detection	31
4.3. Performance Optimizations	31
4.3.1. Perfect Prefetching	31
4.3.2. Branch Outcome Forwarding	32
4.4. Platform Considerations	34
4.5. Implementation Challenges	34
4.5.1. Two Checkpoints for the Leading Core	34
4.5.2. Transaction Length Synchronization	35
4.5.3. Guaranteeing Transaction Commits	35
4.6. Summary	36
5. Single-Threaded Evaluation	37
5.1. Methodology	37
5.1.1. Implementation	38
5.1.2. Stride Prefetcher	39
5.1.3. Limitations	40
5.1.4. Benchmarks	41
5.2. Power Efficiency Evaluation	43
5.3. Summary	47
6. Multi-Threaded Fault Tolerance	49
6.1. Challenges in Multi-Threaded Fault Tolerance	50
6.1.1. Execution Order on Leading and Trailing	50
6.1.2. Input Synchronization	50
6.1.3. Thread Synchronization	52
6.2. Multiversioning	52
6.2.1. Concept	52
6.2.2. Operations	54
6.2.3. Version Metadata	55
6.2.4. Version Management	56

6.3. Transaction Conflicts	58
6.3.1. Fundamentals	58
6.3.2. Detection	58
6.3.3. False Sharing	59
6.4. Fault Tolerance	60
6.4.1. Association of Leading and Trailing Cores	61
6.4.2. Required Version Count	62
6.4.3. Comparison	62
6.4.4. Rollback	64
6.5. Summary	66
7. FPGA Implementation	67
7.1. Multiversioning Implementation	68
7.1.1. Memory Hierarchy	68
7.1.2. Concept	70
7.1.3. Data Structures	72
7.1.4. Hardware Overhead	75
7.1.5. Conflict Detection and Commit	76
7.2. MicroBlaze Interaction	76
7.2.1. Transaction Control	77
7.2.2. Transaction Bounds	78
7.2.3. Configuration	79
7.2.4. Register Backup	80
7.2.5. Transaction Abort	82
7.2.6. Trailing Execution	84
7.2.7. Performance Counters	84
7.3. Devices	87
7.3.1. Memory Controllers	87
7.3.2. UART	87
7.3.3. Debuggers	88
7.4. Optimizations	88
7.4.1. Validation of Automatic Transactions	88
7.4.2. Bloom Filter	89
7.4.3. Cache Line Compression	90
7.4.4. Fresh Fetch	90
7.4.5. Sticky Trailing Threads	91
7.4.6. Important Writes	91
7.5. Summary	92
8. Porting the PARSEC Benchmarks	93
8.1. Bare Metal Execution	94
8.2. Launcher Tool	96
8.3. Pthreads Implementation	97
8.3.1. Thread	97

8.3.2.	Mutex	98
8.3.3.	Barrier	100
8.3.4.	Once	101
8.3.5.	Scheduling	102
8.3.6.	Condition Variable	102
8.3.7.	Thread Local Storage	102
8.3.8.	Other	103
8.4.	Atomic Operations	103
8.4.1.	Atomic Operations in Transactions	104
8.4.2.	Conversion of Atomic Operations	104
8.4.3.	Fallback for Uncaught Load Linked/Store Conditional	106
8.5.	Benchmark Details	107
8.6.	Summary	107
9.	Multi-Threaded Evaluation	109
9.1.	Methodology	110
9.2.	Execution Time Overhead	111
9.2.1.	General Results	111
9.2.2.	Benchmark canneal	113
9.2.3.	Benchmark streamcluster	114
9.2.4.	Race Conditions	115
9.2.5.	Optimizations	116
9.3.	Error Detection Latency	119
9.4.	Fault Injection	120
9.5.	Summary	123
10.	Related Work	125
10.1.	Hardware-Based Approaches	126
10.1.1.	Lockstep	126
10.1.2.	TMR: Triple Modular Redundancy	126
10.1.3.	Diva: Dynamic Implementation Verification Architecture	127
10.1.4.	AR-SMT: Active-stream/Redundant-stream Simultaneous Multithreading	127
10.1.5.	Slipstream Processor	128
10.1.6.	ReStore: Symptom-Based Soft Error Detection in Microprocessors	128
10.1.7.	Transient fault detection via simultaneous multithreading	129
10.1.8.	SRTR: Simultaneously and Redundantly Threaded processors with Recovery	129
10.1.9.	FaultTM-multi	130
10.2.	Software-Based Approaches	131
10.2.1.	PLR: Process-Level Redundancy	131
10.2.2.	SWIFT: Software Implemented Fault Tolerance	132
10.2.3.	HAFT: Hardware-Assisted Fault Tolerance	132

10.2.4. COTS Fault Tolerance with Intel TSX	133
10.3. Comparison	134
10.4. Summary	136
11. Summary and Conclusion	139
11.1. Summary	139
11.2. Future Work	141
11.3. Conclusion	143
List of Figures	xvii
List of Tables	xxv
List of Source Code Listings	xxvii
Bibliography	xxix
A. Bare Metal API	xxxvii
A.1. File System	xxxvii
A.2. Memory Allocation	xxxix
A.3. Compiler Helper	xl
A.4. Various	xl
B. Necessary Changes to the Parsec Benchmarks	xli
B.1. blackscholes	xlii
B.2. bodytrack	xlii
B.3. canneal	xliii
B.4. dedup	xliii
B.5. facesim	xliv
B.6. ferret	xlv
B.7. fluidanimate	xlv
B.8. freqmine	xlvi
B.9. raytrace	xlvi
B.10. streamcluster	xlvi
B.11. swaptions	xlvii
B.12. vips	xlvii
B.13. x264	xlviii

List of Abbreviations

API	Application Programming Interface
AXI	Advanced eXtensible Interface
AXIS	Advanced eXtensible Interface Stream
BRAM	Block Random-Access Memory
CAS	Column Address Strobe
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
COTS	Commercial Of The Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DMR	Dual Modular Redundancy
DSP	Digital Signal Processing
DUE	Detected Unrecoverable Error
ECC	Error Correction Code
FIFO	First In - First Out
FIT	Failure In Time
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
ILA	Integrated Logic Analyzer
IP	Intellectual Property
JTAG	Joint Test Action Group
LMB	Local Memory Bus
LUT	Look-Up Table
HTM	Hardware Transactional Memory

MSR	Machine Status Register
MMU	Memory Management Unit
MTTF	Mean Time To Failure
PC	Program Counter
RISC	Reduced Instruction Set Computer
SDC	Silent Data Corruption
SDK	Software Development Kit
SEU	Single-Event Upset
SMT	Simultaneous Multi-Threading
SRAM	Static Random-Access Memory
SRL	Shift Register Logic
STM	Software Transactional Memory
TCP/IP	Transmission Control Protocol/Internet Protocol
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver/Transmitter
URAM	Ultra Random-Access Memory
USB	Universal Serial Bus
VIO	Virtual Input/Output

1

Introduction

Table of Contents

1.1 State of the Art	2
1.2 Tight and Loose Coupling	3
1.3 Multi-Threaded Execution	5
1.4 Our Solution	6
1.5 Outline	7
1.6 Funding and Publications	8

Even on general-purpose computers, a crash or wrong output is already annoying and disruptive. On embedded systems, however, failure can be significantly more critical. For example, issues with safety-critical applications like autonomous cars or fly-by-wire electronic flight control systems can directly endanger human lives. Therefore, a high reliability is essential to ensure correct behavior at all times. Radiation or power fluctuations can cause flip-flops to assume the wrong value. These bit-flips can further propagate and can cause visible errors in the form of wrong outputs or crashes. Therefore, a form of fault tolerance, which can detect the occurrence of faults before they cause damage, is required. In the case of fail-operational systems, recovery is performed, which enables the system to continue its operation without impact on the safety of its users.

Fault tolerant systems, which are available for purchase today, aim primarily at applications with low computational requirements like airbags or electronic stability control. However, modern safety-critical applications like autonomous cars or advanced terrain awareness and warning systems require both a high performance and high reliability. These systems often employ multi-cores to reach the required performance. Heterogeneous multi-cores, which consist of fast and energy-efficient cores executing the same instruction set, have also gained in popularity. The combination

of these requirements, high reliability, high performance and high energy efficiency, is challenging to achieve with state of the art processors. This thesis proposes a novel approach to fulfill these requirements.

1.1. State of the Art

Fault tolerant systems, which are available for purchase today, primarily rely on lockstep redundancy. Lockstep systems execute each instruction multiple times on different cores synchronously. It is possible to use pairs or triplets of cores for this purpose. The cores' state is compared after each cycle and an error is signaled if they disagree.

The ARM Cortex-R [10] and Infineon AURIX [66] processors are popular representatives of DMR (Dual Modular Redundancy) lockstep systems, which execute each instruction on two cores. However, a major disadvantage of DMR systems is that they cannot determine the faulty instance in case of an error, as both results differ and there is no deciding factor. As safety-critical systems have to continue even when faults are present, an alternative recovery solution is required. Checkpointing, which copies the entire architectural state to safe storage, is a possibility. However, this creates more overhead in addition to the doubling of chip resources and power consumption caused by the replication of the cores. It is possible to perform checkpointing sparingly to optimize performance in the error-free case. On the other hand, this increases the overhead when a fault occurs, as more progress is lost. Additionally, the checkpointing solution itself can be susceptible to errors if it is implemented in software.

TMR (Triple Modular Redundancy) lockstep systems like the Stratus ftServer [59] or the Xilinx MicroBlaze [73] with TMR Voter [74] offer a different solution to recovery. They feature three redundant instances. This enables a majority vote in case of an error. The faulty core is singled out and either disabled or recovered from the state of the other cores. Therefore, such a system can continue execution in the presence of faults. However, adding a third core further increases the cost in terms of chip area and power consumption.

If no suitable fault tolerant processor is available, software fault tolerance remains as the only option. Such systems exhibit more vulnerable parts, since only the redundant application is within the sphere of replication, where it is protected from errors. The mechanisms for error detection and recovery are limited by hardware constraints, which can lead to an inoperable system or data loss. Additionally, software-based fault tolerance often suffers from a higher performance overhead than hardware-based approaches. [44, pp. 297-299]

1.2. Tight and Loose Coupling

Lockstep approaches are tightly-coupled. Instructions are executed either in the same cycle on all cores [44, p. 207] or with a small static delay, which avoids common mode failures [9, p. 11]. However, tight coupling is restrictive. When fault tolerance is engaged, both cores have to be in the exact same state. Uninitialized memory, for example in the branch predictor, is not acceptable, as different branch predictions can lead to divergence, which is detected as error. Inputs have to be delivered to all cores identically and synchronously. This exact input duplication and the per-cycle comparison requires a high bandwidth between cores. Data transfer for duplication or comparison must not be delayed, otherwise the cores might diverge or fail comparison. Consequently, dedicated signal wires between the redundant cores are required. Connecting every core to every other core is very expensive. Therefore, only pairs of cores are connected. This restricts the allocation of redundant pairs, especially when some cores are inoperable.

In heterogeneous multi-cores, pairings of different cores are difficult to realize. The fast core has to be throttled to the speed of the slow core, since comparison is done cycle by cycle. Therefore, the fast core is also not allowed to use a more advanced branch predictor or execute instructions in a different order than the slow core. However, the increased energy consumption, which results from the higher complexity of the fast core, remains. For these reasons, tight coupling of heterogeneous cores is very inefficient.

Loose coupling supports a dynamic delay, called slack, between the two executions. Comparison uses only the architectural state. Therefore, the restrictions of tight coupling are lifted. Branch mispredictions only result in a small fluctuation of the slack instead of a signaled error. Inputs, which are delivered at different times, also only affect timing as long as the values are identical. Loose coupling usually suffices with more coarse comparisons like checksums over multiple cycles. Therefore, the required bandwidth between the cores is reduced and regular memory busses can be used for the input duplication and comparison required for fault tolerance. This permits an arbitrary combination of cores, which is especially advantageous, when some cores are inoperable.

Loose coupling is well suited for heterogeneous multi-cores. Cores can use different speculative mechanisms, as long as the resulting architectural state is identical. The fast core can even assist the slow core by providing hints about cache misses or branch outcomes.

Figure 1.1 illustrates the advantage of loose coupling over tight coupling on heterogeneous multi-cores. In practice, this performance advantage depends on the application and can be even larger than depicted. Optimally, the application offers enough possibilities for the fast core to get sufficiently ahead that the slow core can resolve the entire cache miss before it encounters the triggering instruction. However, cache

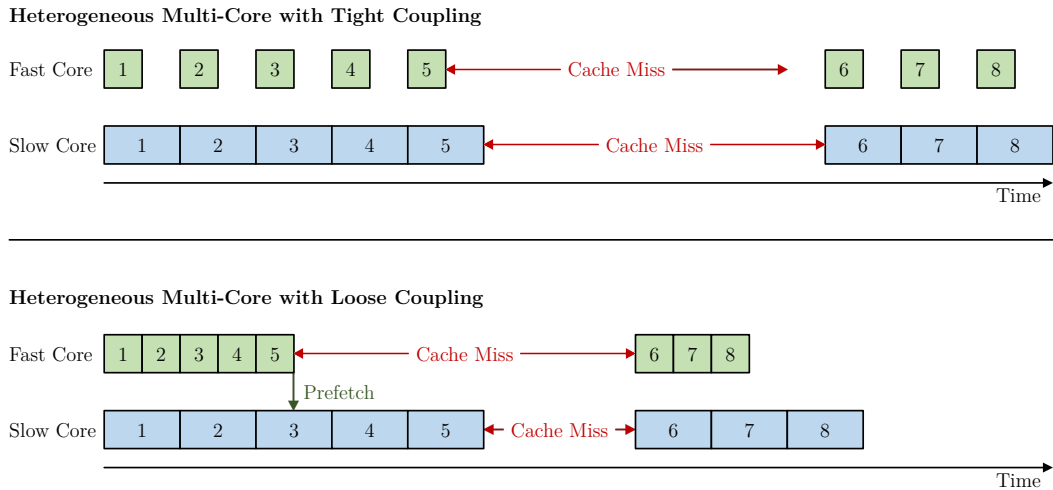


Figure 1.1.: A sequence of eight instructions is executed redundantly on a fast and a slow core of a heterogeneous multi-core. If tight coupling is used, the fast core has to wait after each instruction for the cores to stay synchronous. Similarly, it has to wait for the slow core to process the cache miss. However, if loose coupling is used, the fast core can run ahead of the slow core. When the cache miss occurs, it can issue a prefetch on the slow core. Therefore, the time the slow core spends waiting for the cache miss is reduced. Finally, both cores finish the sequence before their counterparts on the tightly-coupled system do.

misses have to be frequent enough that the fast core does not get too far ahead. Note that a heterogeneous multi-core is not required to profit from loose coupling. For example, a homogeneous multi-core requires less cores for redundancy, when the cores, which are running behind, can be sufficiently accelerated to validate multiple cores, which are running ahead.

Loose coupling offers an advantage at all scales of slack:

- Varying execution orders, which can be caused by an out-of-order architecture on the fast and in-order architecture on the slow core, cause fluctuations in the range of single cycles.
- Different branch predictions cause pipeline flushes on the mispredicting core and fluctuations in the tens of cycles.
- Cache misses can cause fluctuations in the hundreds of cycles.
- Blocking due to synchronization, e.g. due to barriers, can cause fluctuations in the thousands of cycles.
- Dynamic scheduling of tasks can cause fluctuations in the millions of cycles.
- Dynamic execution of processes, e.g. when redundancy is only used in case of spare computing time, can even cause fluctuations in the billions of cycles.

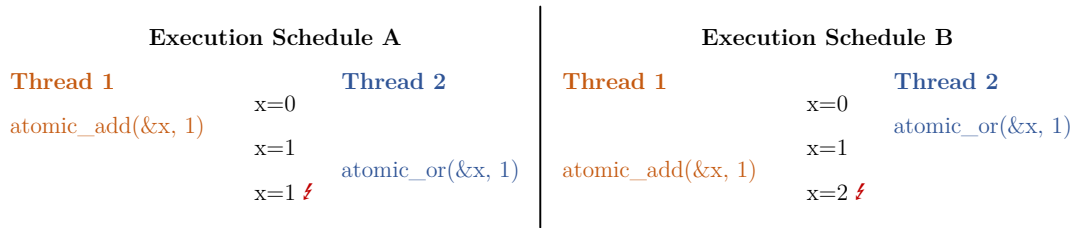


Figure 1.2.: The same application is executed in both execution schedules: Thread 1 increments x by one, while Thread 2 calculates the bitwise OR of x and one. Both operations are performed atomically without explicit critical sections. In Execution Schedule A, the final value of x is one, while it is two in Execution Schedule B. Both results are valid for the given application. However, a simple comparator-based fault tolerance approach falsely detects an error if the redundant instances happen to execute different schedules.

In this thesis, we exclude the extremes. Some approaches to fault tolerance (e.g. AR-SMT [53], SRTR [63]) use the speculative resources of an out-of-order processor combined with additional hardware queues to support a small slack. However, these speculative resources are very limited and the resulting slack is tiny in comparison to our approaches or software approaches. Therefore, we still consider such approaches tightly-coupled. We also do not consider extremely high slacks, as output has to be delayed until both executions have finished. Embedded systems often control physical components, which continue moving during redundant execution. Therefore, they cannot tolerate a high output latency, which makes fault tolerance approaches with extremely high slacks unsuitable.

1.3. Multi-Threaded Execution

Redundant multi-threaded execution is challenging. The memory operations of other cores can cause divergences by changing cache states at arbitrary times. Input duplication also becomes more difficult, as other cores can change memory value at arbitrary times. Therefore, the caches are often not replicated in tightly-coupled multi-cores. This reduces the coverage of the fault tolerance approach and potentially decreases performance, since signal delay increases, as all redundant cores have to access the same L1 cache.

Loosely-coupled systems are not free from issues either. As timing fluctuations are allowed, the execution schedule of the redundant instances can vary. Figure 1.2 shows an example of two different schedules. Both schedules are valid, but result in different values, which is detected by the fault tolerance approach and signaled as an error. Therefore, it is necessary either to restrict schedules in a way that such divergences cannot happen or to extend input duplication so that the result still matches.

A consistent view of memory alone is not sufficient for the execution of multi-threaded applications. A mechanism that enables synchronization of the threads is also required. On non-redundant systems, atomic operations are usually used for this purpose. On redundant systems, these require identical execution schedules. Furthermore, their correct function in redundant execution has to be ensured. For example, an atomic increment is not allowed to increase the target value by two instead of one, when it is executed twice redundantly. Support for additional synchronization mechanisms like transactional memory is advantageous. However, these also require special care. For example, a transaction running redundantly in an unmodified transactional memory would constantly conflict with itself, as it keeps writing identical addresses.

1.4. Our Solution

We propose a novel loosely-coupled fault tolerance approach for heterogeneous multi-cores. Existing transactional memory is used for input duplication and recovery. The conflict detection is adapted for redundant execution. We automatically split execution into transactions, which are executed twice on different cores. Checksums are formed over all instruction outcomes and used for error detection, which reduces the required communication bandwidth. The fast core, hence called leading core, can run ahead of the slow core, called trailing core. Branch outcomes and the addresses of cache misses are transferred from the leading core to the trailing core to accelerate its execution. Therefore, performance is better than what the slow core can achieve on its own, while consuming less power than a system consisting of two fast cores.

We extend our approach to support multi-threaded applications. We resolve the issue of different execution schedules by augmenting the transactional memory to support multiple versions. This multiversioning enables the trailing core to read the exact same values as the leading core even if the execution schedule of the trailing cores differs from the leading cores. The transactional memory's conflict detection is left intact. Therefore, it can be used for synchronization. We tolerate slacks so large that even the wait times before critical sections can be masked. Simultaneously, the error detection latency is short enough for use in embedded systems.

The single-threaded variant was evaluated by executing microbenchmarks on a heterogeneous multi-core in the gem5 simulator [16], which was extended to support our approach. We examined both performance and power consumption. In the best case, our approach offers up to three times the throughput of a lockstep system consisting of power-efficient cores with the same total power consumption. Alternatively, it consumes up to 35% less power than a lockstep system consisting of fast cores with the same throughput.

Our multiversioning approach was implemented on an FPGA (Field-Programmable Gate Array) using the closed-source MicroBlaze [73] cores and evaluated using the

PARSEC benchmark suite [15]. We measured the performance and error detection latency. The mean slowdown of 2.16 compared to the non-redundant baseline is within the expected range for a redundant approach, which executes the application twice. Performance can be enhanced further by optimizing the benchmarks for our platform. The average error detection latency of 9,335 cycles is considered acceptable for most embedded systems. A fault injection analysis demonstrates that our approach could detect and recover from all injected faults.

Altogether, our approach has the following six advantages:

- The system is fail-operational, as it can recover from errors.
- Homogeneous and heterogeneous multi-cores are supported.
- Shared memory multi-threaded applications can be executed redundantly.
- Transactional memory can be used for synchronization.
- No modifications to the cores are required.
- The error detection latency is suitable for use in embedded systems.

1.5. Outline

This thesis is structured as follows: Chapter 2 provides a basic overview of fault tolerance, transactional memory and FPGAs. Chapter 3 describes our hardware architecture, the goal and the scope of our approach. Chapter 4 specifies our single-threaded approach to fault tolerance. The required enhancements to transactional memory, performance optimizations, platform considerations and implementation challenges are detailed. Chapter 5 contains the performance and power efficiency evaluation. Chapter 6 marks the beginning of the multi-threaded approach. It describes the challenges in multi-threaded fault tolerance, our novel multiversioning approach, the handling of transactional conflicts and how fault tolerance can be implemented with multiversioning. Chapter 7 outlines the implementation of an FPGA prototype for our multiversioning approach. It specifies how multiversioning can be implemented, the interaction with the closed-source processor cores, additional devices and optimizations. Chapter 8 states the requirements to port the PARSEC benchmark suite to our prototype. It contains details about the bare metal execution, our launcher tool, required API (Application Programming Interface) functions, our implementation of the pthreads library and atomic operations. Chapter 9 finishes the multi-threaded part with an evaluation. It consists of an evaluation of execution time overhead, optimizations, error detection latency and fault injection. Chapter 10 provides an overview of related work in the field of fault tolerance. Finally, Chapter 11 concludes this thesis with a summary and outlook to future work. Appendix A contains additional information about our implementation of the bare

metal API. Appendix B describes the changes to the PARSEC benchmarks in more detail than the main part.

1.6. Funding and Publications

This thesis is part of the project “Design of Hardware Transactional Memory for Usage in Embedded Systems” (UN 64/19-1), which received funding by Deutsche Forschungsgemeinschaft (DFG).

The approaches presented in this thesis have already been released in shortened form:

Rico Amslinger, Sebastian Weis, Christian Piatka, Florian Haas, and Theo Ungerer. “Redundant Execution on Heterogeneous Multi-cores Utilizing Transactional Memory”. In: *International Conference on Architecture of Computing Systems (ARCS)*. Springer. 2018, pp. 155–167. DOI: [10.1007/978-3-319-77610-1_12](https://doi.org/10.1007/978-3-319-77610-1_12)

Rico Amslinger, Christian Piatka, Florian Haas, Sebastian Weis, Theo Ungerer, and Sebastian Altmeyer. “Hardware Multiversioning for Fail-Operational Multithreaded Applications”. In: *32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2020, pp. 20–27. DOI: [10.1109/SBAC-PAD49847.2020.00014](https://doi.org/10.1109/SBAC-PAD49847.2020.00014)

Rico Amslinger, Christian Piatka, Florian Haas, Sebastian Weis, Theo Ungerer, and Sebastian Altmeyer. “Multiversioning Hardware Transactional Memory for Fail-Operational Multithreaded Applications”. In: *Journal of Parallel and Distributed Computing* (in review 2021). ISSN: 0743-7315

2

Background

This chapter provides an overview of the concepts and technologies that are used in this thesis. A general description of the terminology, classification and remedies of faults is provided. Furthermore, this chapter contains an introduction to transactional memory. An outline of FPGAs and MicroBlaze soft cores is given.

Table of Contents

2.1	Fault Tolerance	10
2.1.1	Faults and Errors	10
2.1.2	Persistence of Faults	10
2.1.3	Impact of Errors	11
2.1.4	Measurement of Reliability	11
2.1.5	Kinds of Redundancy	11
2.1.6	Error Correction Codes	12
2.1.7	Sphere of Replication	12
2.2	Transactional Memory	13
2.2.1	Programming Model	14
2.2.2	Conflict Detection	14
2.3	FPGA and MicroBlaze	15
2.3.1	FPGA	15
2.3.2	MicroBlaze	18
2.3.3	Connectivity	19
2.4	Summary	20

2.1. Fault Tolerance

In practice, no system works flawlessly under all conditions. This section classifies the categories of faults and their impact on the system. The various ways to increase reliability are also outlined.

2.1.1. Faults and Errors

A fault is a defect in either hardware or software [38, p. 2]. Faults can have different forms, like software bugs, manufacturing defects or part wear [44, p. 6]. Undesirable interactions with environmental influences, like radiation, cosmic rays, inappropriate temperature or voltage, are also considered faults [44, pp. 6, 20]. However, a fault does not necessarily manifest, e.g. the broken software might never be run. Additionally, faults are often masked at intermediate layers [44, p. 6].

“Errors are manifestation [sic] of faults.” [44, p. 7] Reasoning about errors depends on the considered scope. For example, corrupted read data is an error at memory level. However, when considering the whole processor, it might not be regarded as one if the memory is part of a branch predictor and the error is so insignificant that it cannot be observed outside of the branch predictor. [44, pp. 7-8]

Errors can propagate through the different layers if they are not masked. Ultimately, they become visible to the user. Once a system no longer meets its correctness or performance guarantees, the term system failure is used. [44, p. 8]

2.1.2. Persistence of Faults

Based on their persistence, faults can be divided into three categories [44, p. 6]:

Permanent faults start occurring some day and continue to occur until the damaged component is repaired. Manifestations of permanent faults are called hard errors. [44, pp. 6-8]

Intermittent faults occur randomly and then seemingly go away. However, if the intermittent fault has arisen once, the likelihood of it emerging again is high. Intermittent faults often turn into permanent faults after some time. [44, p. 6]

Transient faults also occur randomly. Contrary to intermittent faults, the probability of the fault appearing again is not increased after it has occurred once. Manifestations of transient faults are called soft errors. [44, pp. 6-8]

This thesis focuses on transient faults. However, by counting the occurrences of faults on the different cores, the same mechanisms can also be used to detect permanent and intermittent faults.

2.1.3. Impact of Errors

The impact of an error on the user is differentiated [44, p. 3]:

A DUE (Detected Unrecoverable Error) occurs when the system has detected the error, but has no means to recover from them [44, pp. 3-4]. In most cases, a crash is initiated to limit the impact of the error and prevent further data corruption [44, pp. 3-4].

SDC (Silent Data Corruption) eventually causes relevant outputs to be incorrect. The corruption can also spread and cause a crash at a later time. SDC is often considered worse than DUEs, as the incorrect output can go unnoticed and the corruption can also damage the file on persistent storage. [44, pp. 3-4]

Benign errors do not affect the user in any meaningful way, since the system was able to recover from them. However, they are often reported to the user, as they can be an early warning sign for permanent system failure. [44, p. 4]

2.1.4. Measurement of Reliability

The FIT (Failure In Time) rate is commonly used to assess the reliability of a system. It refers to the number of errors in one billion hours of operation. You can estimate the FIT rate of a composite system by adding up the FIT rates of the individual components. However, the true FIT rate is often lower as additional masking can occur or faults in the components might share a single trigger. Fault tolerance mechanisms also require additional consideration when calculating the FIT rate of the complete system. The MTTF (Mean Time To Failure) is an inversely related metric, calculated as $MTTF = 10^9 \text{h} / \text{FIT}$. Its meaning is more intuitive, as it can be interpreted as the expected time until an error occurs. [44, pp. 9-11]

The goal of most fault tolerance approaches is to minimize the FIT of the whole system. However, some approaches only consider SDC as problematic and accept a higher rate of DUE.

2.1.5. Kinds of Redundancy

Generally, there are three approaches to detect or mitigate errors by means of redundancy [58, p. 19]:

Physical redundancy: The component is replicated multiple times in the same system. For DMR (Dual Modular Redundancy) it is replicated twice, which enables error detection by comparing the outputs or state of the copies. In general, one cannot decide which copy is faulty. Therefore, errors can only be detected but not corrected. TMR (Triple Modular Redundancy) adds a third

copy, which enables a majority vote in case of a mismatch. Therefore, execution can continue correctly after an error has occurred. [58, pp. 19-22]

Temporal redundancy: The application is executed multiple times consecutively and its outputs are compared. For error correction the application can be repeated additional times, until the desired number of matching outputs is achieved. [58, p. 22]

Information redundancy: Redundant bits are added to each data word. Depending on how this is done, errors can be detected and even corrected. [58, pp. 22-25]

2.1.6. Error Correction Codes

ECCs (Error Correction Code) are a kind of information redundancy, which can detect and potentially correct bit-flips in stored or transferred data. In a process called encoding, a code translates data words to codewords. The reverse process is called decoding. “The Hamming distance between two codewords is the number of bit positions in which the two words differ.” [38, p. 56] For ECCs codewords are longer than the corresponding data words. Therefore, not all possible bit sequences exist as codewords and the minimum Hamming distance of the code can be greater than one. [38, pp. 56-57]

If the minimum Hamming distance is two, the code can detect all single bit-flips, as there are no valid codewords, which can be reached by flipping a single bit in a valid codeword [44, p. 164]. An example of such codes with Hamming distance two are the parity codes [44, p. 168]. If the minimum Hamming distance is three, the code can detect up to two bit-flips [44, p. 166]. Alternatively, it can correct a single bit-flip by selecting the only codeword, which only differs by a single bit from the corrupted one [44, p. 165]. However, a codeword corrupted with two bit-flips might be replaced with the wrong codeword and the double bit-flip can no longer be detected. An example of such a code is the Hamming code [28]. If the minimum Hamming distance is four, the code can correct a single bit-flip, while still detecting double bit-flips [44, p. 166]. An example of such a code is the Hamming code with parity [28].

2.1.7. Sphere of Replication

The sphere of replication spans the part of a system that is protected by a fault tolerance approach. Therefore, a fault is detected when it reaches the bounds of the sphere of replication at the latest and cannot propagate past it. Depending on the approach, the sphere of replication can be described physically, e.g. in terms of pipeline stages or caches, or logically, e.g. in terms of processes or instructions. Usually, all state within in the sphere of replication is replicated. Therefore, input,

which enters the sphere of replication, needs to be duplicated, while output needs to be compared. [44, pp. 208-212]

Most fault tolerance approaches do not cover the whole system. Some approaches focus on single components to provide an optimized solution, while others cover large subsystems to reduce comparison overhead. Reasoning about the sphere of replication helps to decide, which approaches to combine and where alternate mitigation mechanisms, e.g. at transistor level, are required. [44, pp. 208-212]

2.2. Transactional Memory

The gain in single-threaded performance has slowed down in the last decade, as the possible increase in frequency is limited by power consumption and cooling. Parallel execution offers a solution to further increase performance, but multi-threaded programming is challenging. The developer needs to consider all simultaneous data accesses, as otherwise race conditions can occur. Due to changing schedules, these race conditions are often nondeterministic, making them difficult to debug. Synchronization, for example with mutexes, offers a solution, but often decreases performance significantly.

Databases offer a convenient solution in the form of transactions, which can be executed simultaneously. Each transaction consists of operations, which should be considered as a logical unit. Transactions in databases feature the ACID properties [27]:

Atomicity: Either all operations of a transaction are reflected in the database or none.

Consistency: Successful transactions leave the database in a valid state.

Isolation: Operations inside a transaction are invisible to concurrent transactions.

Durability: Once a transaction has committed, its changes have to be permanent even in case of subsequent failures.

These properties are not only desirable in databases, but also in parallel programming in general. However, the durability property is often ignored, as most parallel algorithms operate purely in volatile memory and can therefore never survive system crashes. Transactional memory is a generalization of the concept for general-purpose programming, in which the operations affect main memory instead of database tables. For transactional memory, the durability property is often replaced by serializability [30]. Serializability states that for every parallel execution a sequential schedule of transactions exists, which results in the same final state [29, p. 25].

2.2.1. Programming Model

Transactional memory can be implemented as HTM (Hardware Transactional Memory) [29, p. 147] or STM (Software Transactional Memory) [29, p. 101]. Hybrid approaches are also possible [29, p. 150]. In STM, function calls and compiler instrumentation is used for the interaction with the transactional memory [29, p. 68]. HTM uses special instructions and extensions of existing instructions [29, p. 148].

Programming transactions is mainly concerned with determining, which operations are considered as a logical unit. First, the transaction is started with a function call or a special instruction. As sometimes a repeated execution of a transaction can be necessary, this also serves as a checkpoint. A rollback can be necessary to ensure consistency. For example, a transaction might read a value twice, which is changed between the reads by a concurrent transaction. These differing read values, could corrupt data structures like linked lists.

Once the transaction is running, the transactional memory system needs to know, which memory operations are part of the transaction. Some transactional memory systems, called implicit, consider every memory operation part of the transaction [29, p. 159]. Contrary, explicit transactional memory systems introduce additional instructions for transactional reads and writes [29, pp. 154-155].

To finish a transaction and commit its changes to make them visible to other threads, another function or instruction is introduced. Additionally, another function or instruction, which can abort the current transaction to return to the starting checkpoint, is often provided. These aborts are useful to interact with non-transactional synchronization constructs like mutexes. Frequently, a check whether a transaction is running is also included, so that these synchronization constructs can decide whether they need to call the abort instruction.

2.2.2. Conflict Detection

To ensure the transactional memory's properties, accesses to the same memory location by separate transactions need to be detected. If at least one of the accesses is a write, a conflict occurs. This conflict is resolved by aborting a transaction, which involves rolling back all changes, and repeated execution of that transaction.

Conflicts can be detected at different levels of granularity. Tracking at byte accuracy ensures that only true conflicts are detected, but involves a high performance overhead. Tracking at a larger granularity, like cache line size, reduces this overhead and enables the reuse of different mechanisms, like cache coherence protocols, for conflict detection. This can lead to the detection of false conflicts, in which separate memory words in one detection unit were accessed. However, this is mainly a performance issue, as the unnecessarily repeated execution will still produce the correct result. [29, p. 22]

Conflicts can be detected at different times. Most commonly, conflicts are detected, either when they occur (eager) or when the transaction commits (lazy) [29, p. 22]. It is even possible to detect them at different times depending of the type of operation. For example, in a cache coherent system the other caches need to be notified of writes anyway, so conflict detection can be performed eagerly at the same time. However, for performance reasons individual reads are not broadcasted. Therefore, a collective check at the commit needs to be performed, which detects conflicts lazily.

When a conflict is detected, it can be resolved immediately by aborting one of the transactions. However, sometimes it can be advantageous to continue the execution speculatively. For example, conflicts can be ignored at first and once a transaction attempts to commit all conflicting transactions are aborted. This approach has the advantage that forward progress is guaranteed, while with immediate aborts two transactions could alternately abort each other forever. [29, pp. 49-50]

The combination of the selected parameters for granularity, conflict detection and conflict resolution determines the performance of the transactional memory system. However, they cannot be freely chosen, as other components can restrict what can be implemented. Other components can also impose further restrictions on the transactional memory. For example, if the cache coherence protocol is used for conflict detection, the size of the transactions cannot exceed the cache size. [29, pp. 19-23, 151-152]

2.3. FPGA and MicroBlaze

We use an FPGA-based development board for our prototype. To avoid the implementation of a new processor, we utilize the Xilinx MicroBlaze soft core.

2.3.1. FPGA

An FPGA (Field-Programmable Gate Array) is a programmable microchip consisting of many CLBs (Configurable Logic Block). FPGAs are commonly used to speed up applications, which do not justify the expensive manufacturing of custom accelerators. The programming information of the FPGA, called bitstream, is generated from hardware description languages in a process, called synthesis and implementation. With some changes, code written for FPGAs can also be applied to integrated circuits. Due to the hardware description's portability and unlimited reprogrammability, FPGAs prototypes can be used as first step in the development of a new microchip.

In this thesis, we use Xilinx nomenclature, as we use a Xilinx FPGA for our prototype. Figure 2.1 shows a schematic overview of a Xilinx FPGA. A CLB [76] is used to

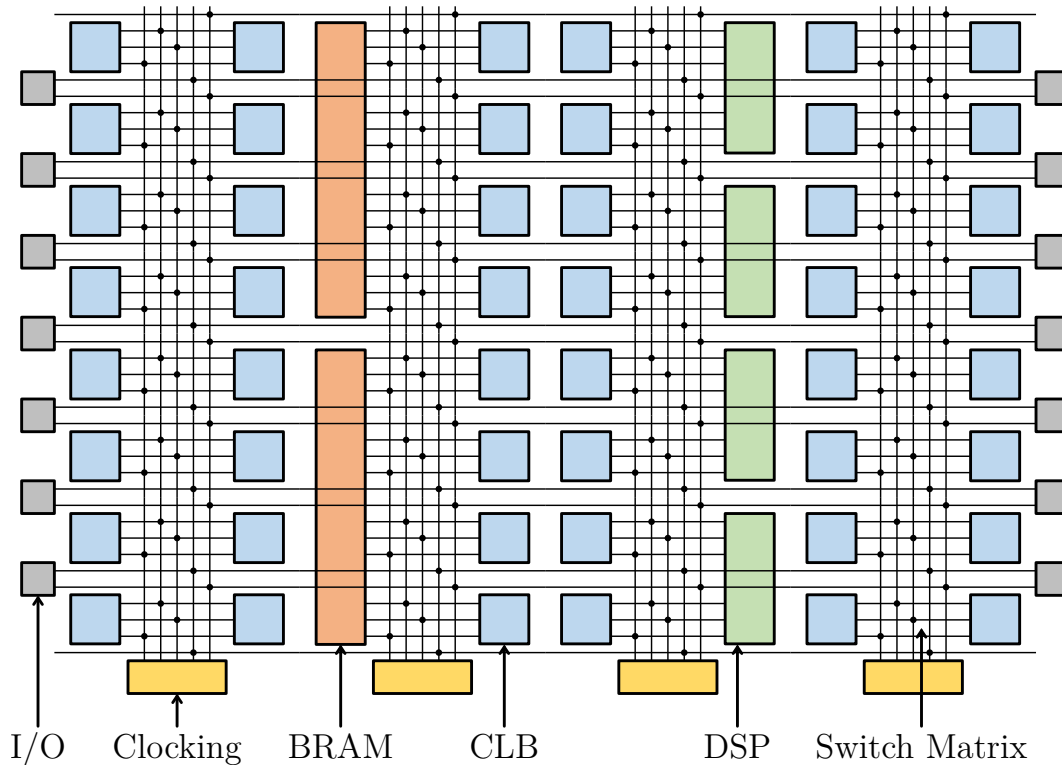


Figure 2.1.: This schematic representation of an FPGA is inspired by the Xilinx Virtex family and its depiction in the tool Vivado. The different components are arranged in repeating rows with larger components like BRAMs and DSPs requiring multiple rows for one instance. Switch matrices provide the connectivity between components. CLBs, BRAMs and DSPs are interleaved, while I/O is located in discrete columns. Multiple rows form a cloning region, which shares its global clocks. However, every column contains additional cloning resources, which can be used to select between them. This schematic is reduced massively in scale. An actual FPGA contains more wires per switch matrix, more components per row, more rows per clock region and also multiple clock regions.

implement a part of a circuit. For this purpose, it consists of multiple programmable components:

- LUTs (Look-Up Table) are small SRAMs (Static Random-Access Memory) used to represent arbitrary combinatorial functions with a limited number of input bits. Their content is set while programming. When the FPGA is in use, the input signals are connected to the address lines, resulting in a lookup of the output value. On Xilinx FPGAs any function with up to 6 inputs and 1 output bit or any function with up to 5 shared inputs and 2 output bits is supported. Additionally, larger functions can be directly represented by combining multiple LUTs in the same CLBs with the hard-wired multiplexers. However, as the size of a LUT grows exponentially with its number of inputs,

splitting the function into subfunctions with fewer inputs is preferred.

- Some CLBs support writing to the LUT's memory at runtime. In addition to fixed combinatorial functions, these blocks can be used as small memory or SRL (Shift Register Logic).
- Each CLB contains dedicated carry logic for arithmetic functions. These can be used to implement addition and subtraction with a lower latency and space requirement than pure LUT logic. It can also represent equality and magnitude comparisons.
- For each LUT output a dedicated storage element is available. These can be configured as flip-flops (registers) or latches. In addition to the data input and output pins, clock, enable and reset pins are available.

Every CLB is connected to a switch matrix [76, p. 54]. A switch matrix is a programmable component, which can establish signal connections. Together they form an interconnect, which makes it possible to connect any driver pin to any sink pin. They also contain internal inverters, which are used to amplify the signal, so it can reach far away pins or fan out to multiple pins.

In addition to these basic components, specialized blocks are available:

- Dedicated memory provides a larger quantity of SRAM than what is available due to LUTs and registers. Xilinx FPGAs contain two different kinds of memory: The smaller BRAM (Block Random-Access Memory), which stores 36 kbit, is highly configurable, while the larger URAM (Ultra Random-Access Memory), which stores 288 kbit, supports fewer configuration options. Importantly, the initial state of BRAM can be set while programming, while URAM is always initialized to zero. [78]
- DSPs (Digital Signal Processing) [77] can be configured to perform various arithmetic operations. Particularly, they can be used to calculate integer multiplications, which would require a large area, when implemented with LUTs.
- I/O tiles make the external pins of the FPGA available to the programmed logic. The voltage and direction of the pins can be configured. Tristate, in which a pin can function as input and output at the same time, is also supported. Hard-wired transceivers can be used for high-speed communication. [79]
- Various components are available to distribute clock signals. They can also alter the frequency or provide phase-shifted or distorted variants of the base clock. [75]

2.3.2. MicroBlaze

The Xilinx MicroBlaze [73] processor is intended to run on Xilinx FPGAs like the one we use for our prototype. It is a soft core, which means it utilizes the FPGA's general-purpose components like CLBs or DSPs. It can be configured as both a 32-bit and 64-bit architecture. However, as memory accesses are more complex in the 64-bit variant, we selected 32-bit for our prototype. A hardware floating point implementation can be enabled, which greatly benefits some benchmarks. In the 32-bit variant only single precision floats are available, while the 64-bit architecture also support double precision floats. The processor can be used in shared memory multi-core architectures, as it supports the load linked/store conditional atomic operations.

Integrated L1 caches are available. However, we disabled them, as we provide our own cache implementation. For memory access, LMB (Local Memory Bus) [71, 73] and AXI (Advanced eXtensible Interface) [41] ports are provided. LMB is an interface to connect processors to high-speed peripherals with low latency that is primarily used by Xilinx IP (Intellectual Property). AXI is a more feature-rich, but also more complex interface by ARM that is used by a wide range of manufactures. Instruction and data ports are separated (Harvard architecture), but the interfaces can be merged at a later point in the memory hierarchy if a suitable linker script is used to compile the applications. Optionally, an MMU (Memory Management Unit) is available. Furthermore, up to 16 AXIS (Advanced eXtensible Interface Stream) interfaces can be enabled.

The MicroBlaze processor uses a custom RISC (Reduced Instruction Set Computer) instruction set. A version of the GCC compiler, which can output these instructions, is provided with the Xilinx SDK (Software Development Kit). There are 32 general-purpose registers, of which the first one is always zero. Floating point instructions share these general-purpose registers. Each instruction is 32 bit long, which restricts immediate operands to 16 bit. However, a special `imm` instruction can extend the immediate operand of the next instruction to 32 bit. This instruction behaves like a prefix in a CISC (Complex Instruction Set Computer) instruction set, as the instruction pair must not be interrupted. Unfortunately, it is not possible to prevent the compiler from emitting the `imm` instruction, which would simplify some low-level operations required for our approach.

The MicroBlaze processor supports a vector table, which contains entries for reset, exceptions, interrupts and breaks. Only a single interrupt pin is provided. Therefore, an external interrupt controller is required if multiple interrupt sources are present. Hardware exceptions can be enabled in groups at design time.

Branches in the MicroBlaze instruction set can optionally use a delay slot. This delay slot following the branch instruction is always executed even if the branch is taken. Conditional branches use a register as condition. Flags, like the carry flag, cannot

be used as condition directly, but have to be loaded in a register first. Some branch variants store the PC (Program Counter) in a register. This is especially useful for function calls, but complicates exception handling, as a branch can change its own condition.

It is possible to debug the MicroBlaze cores using a MicroBlaze Debug Module [72]. The debugger itself runs on a host machine and communicates with the MicroBlaze Debug Module using JTAG (Joint Test Action Group). On modern development boards, the JTAG communication often uses a USB (Universal Serial Bus) port for convenience. The MicroBlaze Debug Module also provides additional features like memory access and virtual UART (Universal Asynchronous Receiver/Transmitter) ports. Each MicroBlaze Debug Module can handle up to 32 cores.

An optional trace interface is also available. This interface provides information about the executed instruction, branches, memory accesses, register writes and pipeline state. All signals are output only and cannot influence the processor state.

The MicroBlaze processor provides integrated support for lockstep operation. Two or more MicroBlaze cores can be configured to execute the same program. Their state is compared after each cycle. A master core handles communication with the memory and devices. Inhibitors prevent the propagation of errors if a state mismatch is detected. Furthermore, cores are halted immediately. Xilinx does not provide a recovery scheme for lockstep execution. However, Xilinx also offers a TMR voter implementation [74], which is compatible with the MicroBlaze core.

2.3.3. Connectivity

Xilinx provides various IPs to support easy communication with the FPGA's environment. The Memory Interface Generator [80] can be used to generate memory controllers for external memories. Particularly, it can address external DDR3 and DDR4 memory, which have rather complex interfaces. On the FPGA-side, an AXI interface is provided.

A UART controller [69] is also available. It also features an AXI interface. Many development boards include an adapter chip to convert the UART port to USB. Alternatively, the MicroBlaze Debug Module provides a compatible AXI interface, but transfers the UART communication over JTAG. This is convenient if a MicroBlaze Debug Module is needed for debugging anyway.

The ILA (Integrated Logic Analyzer) [70] provides additional signal debugging. It provides an integrated digital oscilloscope. Conditions for triggers can be defined in a GUI (Graphical User Interface) on the host system and captured data is transferred over JTAG. If interaction with the logic on the FPGA is also required, a VIO (Virtual Input/Output) [81] can be used. It provides bidirectional communication between the host and the FPGA over JTAG. The separate signals can be accessed with a

GUI or by scripting. However, for larger transfers the MicroBlaze Debug Module is preferable, as it implements AXI, which enables higher transfer speeds than raw signal communication.

If more AXI devices are used than AXI ports are available, an AXI interconnect [68] can be used. On both sides, up to 16 interfaces are supported. An address map is used for routing. The interconnect can translate between different AXI configurations, like varying protocol versions, data widths or clocks. Register slices can be inserted to fulfill timing requirements.

2.4. Summary

Current systems suffer from defects, which affects their reliability. Faults can be classified according to their persistence as permanent, intermittent or transient. They can cause DUEs (Detected Unrecoverable Error), SDC (Silent Data Corruption) or benign errors. Reliable systems require a mechanism to tolerate faults. This can be achieved with physical, temporal or information redundancy. ECCs (Error Correction Code) are a prominent form of information redundancy.

Transactional memory is a synchronization approach inspired by database transactions. It uses conflict detection to guarantee atomicity, consistency, isolation and serializability. An implementation in both hardware and software is possible. Various characteristics, like conflict detection granularity and handling, determine the transactional memory's performance.

FPGAs are reprogrammable components, which can be used to prototype hardware. They consist of a large number of elements such as CLBs (Configurable Logic Block), switch matrices, DSPs (Digital Signal Processing), memories and clocking resources. The MicroBlaze is a soft core developed by Xilinx. Various other IP (Intellectual Property), like a memory controller, ILA (Integrated Logic Analyzer), VIO (Virtual Input/Output) and an AXI (Advanced eXtensible Interface) interconnect, are also available.

3

Execution Model

This chapter describes the baseline hardware architecture and memory hierarchy, which was used in this work. In addition, the fundamental redundancy approach is explained abstractly.

Table of Contents

3.1	Hardware Architecture	21
3.2	Redundant Execution	22
3.2.1	Goal of Our Approach	22
3.2.2	Scope of Our Approach	23
3.2.3	Automatic Transactions	23
3.2.4	Concept	24
3.3	Summary	25

3.1. Hardware Architecture

We assume a shared memory multi-core as depicted in Figure 3.1. The cores can be homogeneous or heterogeneous, but have to execute the same instruction set. All approaches require at least two cores. The cores are connected to coherent private instruction and data caches. Optionally, any number of shared caches can follow. However, they are not needed for the presented approaches to work. Finally, all caches are connected to a shared main memory. Other hardware components (like UART controllers) exist at the same level as the main memory and are mapped into the high half of the address space.

To enable transactional memory or multiversioning support, it is necessary to extend the cores and data caches. If the executed code is not self-modifying, an extension

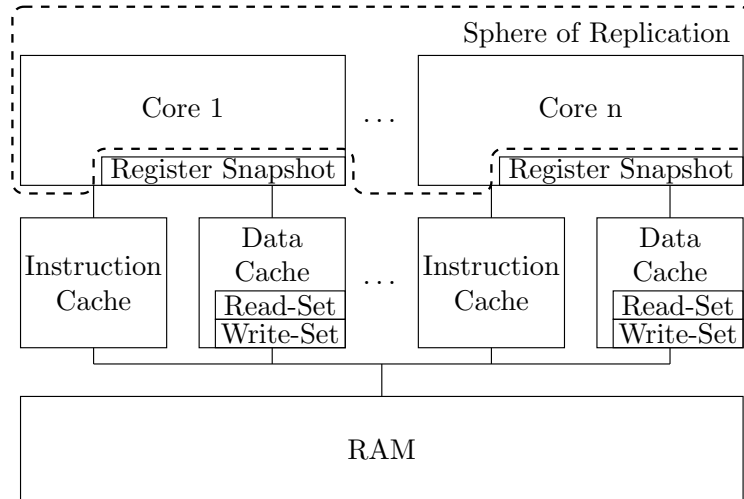


Figure 3.1.: The cores and data caches are extended to support transactional memory and multiversioning. The register snapshot can also be stored external to the core, as its access latency is not performance critical. The sphere of replication, marked by the dotted line, covers the pipeline in the cores. The remaining components are protected by ECC.

of the instruction caches is not required. The cores themselves are extended to store register snapshots. These snapshots are used to roll back in case of a conflict and to recover from faults. The data caches contain the largest part of the transactional memory logic. They are extended to store the read- and write-set of the transactions. In addition, they handle conflict detection, version selection and rollback by themselves. The cores communicate with the data caches by utilizing a memory-mapped interface. The caches notify the cores of conflicts by issuing an interrupt.

3.2. Redundant Execution

3.2.1. Goal of Our Approach

The goal of this approach is to provide a widely useable fault tolerance solution for embedded systems. As a quick fallback to a fail-safe state might not always be possible in modern embedded applications like autonomous cars, the approach has to be able to recover from faults. The power consumption needs to be low, as embedded systems are often powered by battery.

At the same time, it has to offer high performance for complex operations like lane or pedestrian detection. To realize this performance, multi-threaded execution needs to be supported. The support of heterogeneous systems enables favorable tradeoffs between power consumption and high performance. If a system has a low overall

performance requirement, the approach has to work with few cores to save costs and power.

The approach has to support running multiple applications, of which only some might require fault tolerance. If redundancy is not needed for an uncritical application, the redundant cores can be used for higher performance instead. Therefore, maintenance operations need to be provided for an operating system. At the same time, bare metal execution has to be supported for systems dedicated to one purpose and accurate performance evaluation without operating system influences.

3.2.2. Scope of Our Approach

The goal of the presented approaches is to protect the cores' pipelines from transient errors in the form of SEUs (Single-Event Upset). Thus, the sphere of replication (see Figure 3.1) encapsulates them. Applications are protected while they are running in user mode. The execution of the kernel is not protected, as some actions, like powering off the core, cannot be rolled back if an error occurs. Interaction with external devices is also not protected, as this would require them to support rollback, too.

We assume that the remaining memory hierarchy is protected by different approaches like ECC. It is also necessary to protect the cores' register sets to enable rollback to a safe state. If the whole register set is copied to create a snapshot, this can also be realized by ECC. Complete system failure, e.g. by a power failure or unintended global reset, has to be made impossible by technical measures.

3.2.3. Automatic Transactions

As our approach is based on transactions, it is necessary to execute the application in transactions. However, this is not the case for most applications, which are currently in use. Manually converting an application to use transactions everywhere, is laborious and difficult. Transactions are limited in length on most HTM systems, which makes it necessary to insert many boundaries. Statically determining when the allowed length is exceeded is difficult, as it involves analyzing cache misses. At the same time, transactions should not be too small, as this causes a performance loss. Synchronization constructs require additional attention, as writes will only become visible to other cores on commit.

For these reasons, we suggest to use automatic transactions. Contrary to regular transactions, those automatic transactions commit by themselves after they become too large or exceed a given runtime. The next transaction starts immediately afterwards. These automatic transactions cannot be used for concurrency control, as they might commit in the middle of a critical section. However, manual transactions,

whose bounds are set by the programmer, can also be used if needed for concurrency control.

The implementation of automatic transactions depends on the platform. For example, the effect of cache evictions and restricted instructions needs to be considered. Therefore, they will be explained in more detail in Section 4.2.1.

3.2.4. Concept

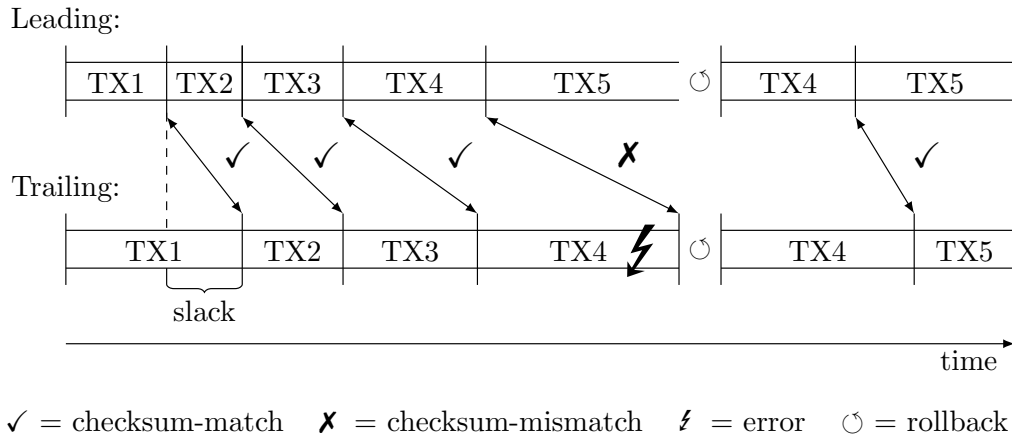


Figure 3.2.: The single-threaded application is split into transactions TX_i , which are executed on a leading core and a trailing core. For some time the checksums match, but after TX_4 a bit-flip causes a mismatch. This results in a rollback and a restart of TX_4 . The delay between the redundant executions is called slack.

The fundamental fault tolerance approach, which is used in this work, is dual modular, loosely-coupled redundancy. The loose coupling is realized by a leading/trailing execution concept (see Figure 3.2). The program is first executed on (a) leading core(s). The results are then validated by (a) trailing core(s), which execute the same code. Depending on implementation details, it can be favorable to begin trailing execution either directly with leading execution or at a later time.

To realize this concept, the execution is automatically split into transactions (TX_i in Figure 3.2). An error cannot propagate to the other core, as the HTM ensures all cores can only see their modifications to the memory.

When and how the trailing transaction is started depends on the approach (single-/multi-threaded). However, both approaches require the leading core's transaction to finish before the corresponding trailing transaction. It can then already start the next transaction, as long as it has sufficient speculative resources. Though, it still has to keep information like old register or memory values of the first transaction, as they are required for rollback if an error occurs. The trailing core is not allowed

to get ahead of the leading core, as this would complicate validation and recovery. In practice, this should rarely happen, though.

While the transaction is running, a checksum of every instruction outcome is calculated. This checksum is then compared after both transactions have completed. If the checksums match, execution can continue regularly and the transactions finalized by wiping the remaining rollback data. If the checksums do not match (after TX4 in Figure 3.2), both cores need to roll back to the beginning of their transactions. This means, the leading core might have to roll back multiple transactions at once. After the rollback, both cores restart their transactions. If the fault was transient, it should not occur again and the checksums should match after both transactions have been repeated.

If a fault occurs, one of the transactions might get stuck. For example, a bit-flip could change the return address to the return instruction itself. To handle these cases, a watchdog timer for each core is needed. The watchdog timer starts, when the first transaction is started. It is then restarted, whenever a transaction commits. If the timer triggers, a fault is assumed and a rollback is initiated. The interval of the watchdog timer has to be large enough that a transaction will always automatically commit before it triggers. The interval also needs to contain buffers for potential blocking. For example, the leading core could be out of speculative resources and waiting for the trailing core to commit its transaction. Alternatively, the watchdog timer can be paused, while the core is blocked. However, it should not be possible for broken software to cause such a pause.

3.3. Summary

This chapter introduced the hardware architecture and the proposed redundant execution concept in general. The assumed memory architecture is a shared memory multi-core with coherent caches. The cores and caches are extended to supported transactional memory or multiversioning. This approach attempts to protect the cores' pipelines from SEUs. All other components are protected by different approaches.

The proposed fault tolerance approach is dual modular loosely-coupled redundancy. This is realized by a leading/trailing execution concept, in which the program is automatically split into transactions, which are executed redundantly. The outcome of these transactions is compared after each commit. If a mismatch occurs, they roll back to the beginning of the transaction. In addition, a watchdog timer is used to detect stuck cores.

4

Single-Threaded Fault Tolerance

This chapter describes the single-threaded variant of the fault tolerance approach. The approach in this chapter was already released in shortened form as [4].

Table of Contents

4.1	Concept	28
4.2	Enhancement of the Transactional Memory	28
4.2.1	Automatic Transaction Bounds	28
4.2.2	Error Detection	30
4.2.3	Double Checkpoints	30
4.2.4	Conflict Detection	31
4.3	Performance Optimizations	31
4.3.1	Perfect Prefetching	31
4.3.2	Branch Outcome Forwarding	32
4.4	Platform Considerations	34
4.5	Implementation Challenges	34
4.5.1	Two Checkpoints for the Leading Core	34
4.5.2	Transaction Length Synchronization	35
4.5.3	Guaranteeing Transaction Commits	35
4.6	Summary	36

4.1. Concept

The approach is based on the leading trailing concept, which was presented in Section 3.2. This variant is optimized for the execution of single-threaded applications on a heterogeneous multi-core. It is possible to execute multiple single-threaded applications in parallel if additional cores are available. It is also possible to disable fault tolerance for applications, which do not need it, as all cores can run software independently.

The goal is to protect the application while running in user mode. Interaction with devices outside of the sphere of replication cannot be protected easily with this approach, as they would need to support rollbacks. Thus, when a system call occurs, the system first waits for the trailing core to catch up to ensure the correctness of the instruction, which causes the system call, itself. Then redundancy is disabled and the operating system handles the system call on the leading core only. Afterwards, the operating system is responsible for synchronizing the trailing core's state to the leading core. Special care needs to be taken, as minor differences in hardware state, which are not even visible in user mode, can result in differing transaction lengths.

4.2. Enhancement of the Transactional Memory

To provide fault tolerance for single-threaded applications, a regular transactional memory can be used. Some transactional memory features like isolation can be reused for fault tolerance unaltered. It is necessary to implement some extensions for the HTM, though.

4.2.1. Automatic Transaction Bounds

To avoid modification of the application, which is intended to be run redundantly, it is necessary for the transactions start and commit automatically. In a regular transactional memory system, the transaction bounds are commonly marked by explicit instructions. However, in our approach, the CPU (Central Processing Unit) and the L1 cache define the bounds by themselves. Figure 4.1 shows an example of these bounds. The approach is described in detail below.

The transaction start is simple to implement. As we intent to protect the whole application, a transaction starts, whenever the processor switches to user mode. A transaction also starts after another one commits or a rollback has concluded to ensure that not a single instruction is executed unprotected.

Determining the optimal timing to commit the transaction is more complicated. One wants to avoid committing too often, as every transaction commit and subsequent start incurs some overhead. At the same time, excessively long transactions should

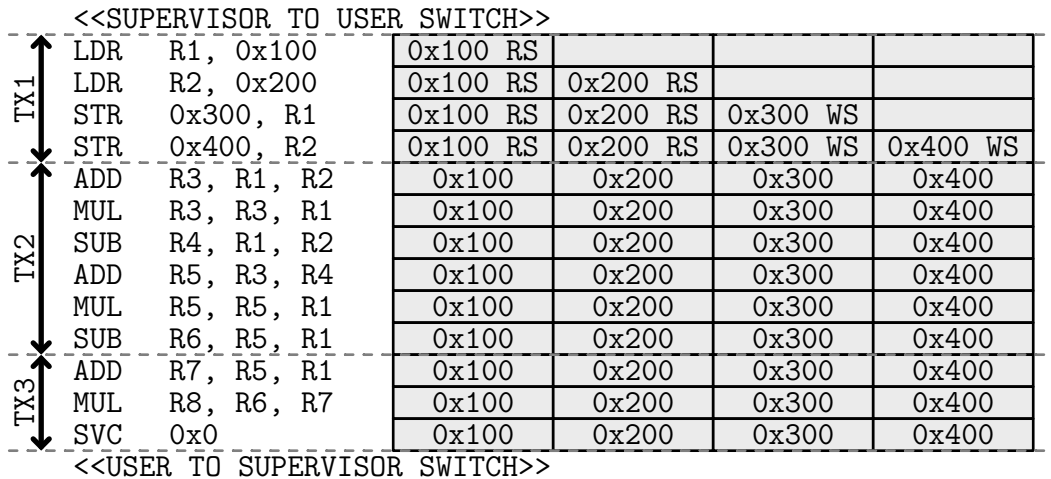


Figure 4.1.: The automatic transaction boundaries of an exemplary execution are shown above. At first, the CPU switches from supervisor to user mode with fault tolerance enabled. This causes the transaction TX1 to start. The program begins by executing four memory accesses. After the last store, the cache is full and every cache line is part of either the read- or write-set. Thus, TX1 commits and TX2 starts immediately. The cache is still full. However, the cache lines are no longer part of the read- or write-set. Hence, they could be evicted and no additional commit is necessary because of them. In the further execution, the program performs some arithmetic operations. To keep the transaction length roughly equal, the system was configured to limit transactions to six instructions. Therefore, TX2 commits after the subtraction and TX3 starts. Finally, the program executes a system call. TX3 commits before the switch to supervisor mode and no further transaction is started, as the kernel is not protected by this approach.

be avoided, too. Many transactional memory implementations already limit the size of a transaction, as they can only use the L1 cache for speculative storage. This can result in very small transactions, especially on caches with low associativity, as the transaction has to end before even a single set overflows. For this reason, the L1 cache should have an associativity of at least eight. To ensure this limitation is met, the cache has to send a signal to the CPU as soon as a set is full.

This approach is easier to implement than determining the exact point where the cache overflows, as only a single case has to be handled. In addition, it is off the critical path, as the current memory operation can always continue. For out-of-order CPUs this might still be difficult to implement, as stores to different addresses in the same set, might have swapped order. This has to be avoided to ensure that both the leading and the trailing core choose the same point to commit their transactions. Other effects, which might cause the caches to fill up at different times, have to be avoided, too. For example, this might happen if the transactional memory system uses a cache line with core-dependent address to store metadata.

If a long transaction is followed by a short transaction, the leading core might run out of speculative resources, as the trailing core might not have finished the long transaction yet, requiring the leading core to keep the checkpoint that was created at the beginning of the first transaction. Thus, transactions should be roughly the same length. Therefore, a static time limit should also be used to determine the commit timing. This also simplifies the implementation of the watchdog timer, as this makes a static interval possible. As both the leading and the trailing core need to commit at the same time, it is easier to express the time limit in instructions executed instead of milliseconds.

4.2.2. Error Detection

The transactional memory system has to be extended to detect errors.

Checksum

One possibility is to compare checksums. These checksums are updated whenever an instruction commits. The major advantage of this approach is that it can be easily implemented on most transactional memory systems. However, a checksum might miss errors if they propagate over multiple instructions. This can be mitigated by enlarging the checksum, as this lowers the probability that a random modification of the checksum results in the other core's checksum. Another disadvantage is that a checksum will also detect many benign faults, as they cannot be masked in the checksum.

Full Comparison

An alternative is to compare every written cache line and all registers. This approach is most suitable for transactional memory systems like [19], which transmit information about the accessed cache lines for conflict detection at commit. These messages are extended to transmit not only the addresses but also the modified data. In addition, all register values are stored at a reserved location, so that they are included in the comparison. This approach detects every faulty bit, which has persisted until commit. However, many benign faults are ignored, as the corrupted values, which are already overwritten at commit time, do not cause a mismatch.

4.2.3. Double Checkpoints

As the trailing core will often lag behind the leading core, the leading core needs to be able to start another transaction, while the last one is not confirmed yet. Thus, the capability to create a second checkpoint is needed. This requires a second register snapshot. In addition, an additional bit is needed for each cache line to mark to which of the two transactions the cache line belongs.

4.2.4. Conflict Detection

Conflict detection is not needed for single-threaded redundant execution. Quite the contrary, it would even hinder redundant execution, as there would constantly be conflicts between the leading and trailing core. Thus, a configuration option to disable conflict detection is needed.

4.3. Performance Optimizations

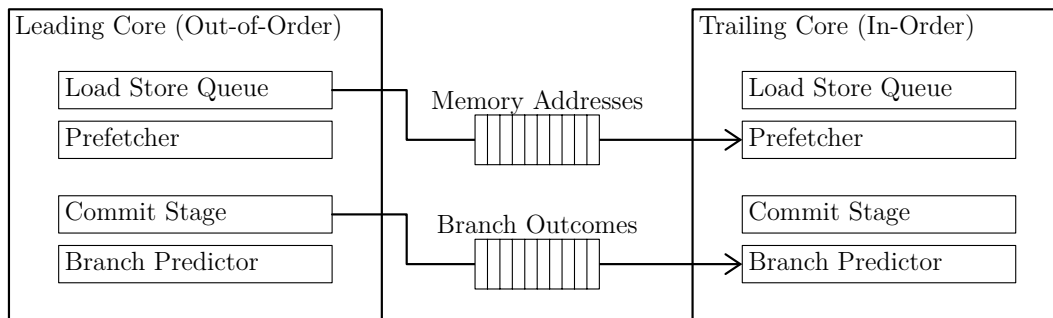


Figure 4.2.: In order to accelerate the trailing core, data is transmitted from the leading core. Perfect prefetching (see Section 4.3.1) uses the memory addresses from the leading core’s load store queue to improve the trailing core’s prefetcher. Branch outcome forwarding (see Section 4.3.2) uses the branch outcomes from the leading core’s commit stage to improve the trailing core’s branch predictor. Both solutions require an additional queue between the cores to cover the slack.

To increase the performance several optimizations can be employed. On a heterogeneous system, it is often sufficient to accelerate the trailing core to observe an increase in total system performance.

4.3.1. Perfect Prefetching

Common prefetchers struggle with data structures like Figure 4.3, which are stored in memory noncontinuously. In our approach, the memory addresses, which were accessed by the leading core, can be forwarded to the trailing core. Those can then be used to prefetch the corresponding cache lines. In the error-free case, this results in all cache misses being eliminated. This is especially advantageous for in-order CPUs, which cannot mask cache misses.

A hardware queue (see Figure 4.2) is used to transfer the addresses from the leading core to the trailing core. Entries are only removed from the queue if they are for the current transaction. As transactions have to fit in the cache, this limitation is sufficient that no cache lines are evicted too early.


```
    }  
  }  
  return concatenate(quicksort(lower), pivot, quicksort(upper));  
}
```

Listing 4.1: This listing contains a recursive implementation of the quicksort algorithm. First a pivot element is selected. The vector is then split into a vector with all elements smaller than the pivot and another vector with all elements greater or equal to the pivot. The algorithm is then applied recursively to both vectors.

The branch predictors, which are used by modern high performance, out-of-order CPUs, are very advanced and can handle most situations with low misprediction rates. However, some branches are impossible to predict without actually executing the condition. For example, the branch, which compares the vector element to the pivot in Listing 4.1, is impossible to predict, when the source data is random. Additionally, leading processor manufacturers do not publish the algorithms behind those advanced branch predictors. Thus, CPUs, which were made by another manufacturer, often use less advanced branch predictors with more mispredictions.

In our approach, the outcomes of branches (taken or not, target address) can be forwarded from the leading to the trailing core. These outcomes can then be used as branch predictions by the trailing core. This improves the performance especially for hard to predict algorithms like sorting.

A hardware queue (see Figure 4.2) is used to transfer the branch outcomes from the leading core to the trailing core. However, additional information is needed to correlate the branches. For example, a complex out-of-order CPU might replace short branches with predicated execution [37]. In this case, the branch itself will never reach the leading core's commit stage and an entry is missing in the branch outcome queue. Thus, the trailing core needs to detect this issue and predict the branch itself. If the cores lose synchronization, performance is decreased, as the shifted branch outcomes are most likely less accurate than the trailing core's regular branch prediction.

This optimization does not affect error detection accuracy. If a fault occurs in the leading core, the trailing core might receive false branch outcomes. However, those are only used for the branch predictor. The control flow in the trailing core will comply with its own outcomes just with additional branch mispredictions. If a fault occurs in the trailing core, it will also result in branch mispredictions, but the control flow will follow the false path, which was computed by the trailing core. Thus, the result of both executions will be the same, as when this optimization was not implemented. Accordingly, the comparison logic will detect the mismatch and start recovery if a fault has occurred in one of the cores.

4.4. Platform Considerations

If one wants to employ this approach, it is recommended to use a heterogeneous multi-core, as the leading core should be faster to run ahead. The approach only supports single-threaded applications, so a dual-core is sufficient. However, a larger multi-core can be used if multiple independent applications are run simultaneously.

To profit from the presented performance optimizations, suitable structures have to be available. Concretely, this means the cores need to employ a branch predictor and prefetcher. This should be standard for current shared memory architectures.

As the leading core has to run ahead, it has to be sufficiently fast. Thus, it is advantageous to employ out-of-order execution for the leading core. The trailing core can be slower. It can use in-order execution to save power instead.

The memory hierarchy should include at least one shared cache level to reduce the memory load. The cache lines accessed by the leading core should still be contained in the shared cache, when the trailing core accesses them. If multiple applications are run at the same time, it might be advantageous to deviate from the usual memory hierarchy of heterogeneous systems. While usually all similar cores share a cache, for this approach instead each pair of a fast and a slow core should share a cache. This reduces the cache miss latency for the trailing core, as the cache line is then contained in a closer cache level.

4.5. Implementation Challenges

There are several challenges when implementing this approach in hardware.

4.5.1. Two Checkpoints for the Leading Core

The leading core needs to have two checkpoints available at the same time. If only one checkpoint is available, the system slows down significantly, as now the leading core has to wait for the slower trailing core to finish its transaction. However, adding a second checkpoint to conventional HTM is very difficult. In a conventional HTM system, there is only one cache line per address in the L1 cache. If it is not marked, it contains the value before the checkpoint. If it is marked, it contains the current, speculative value. The old value is then stored in another level in the memory hierarchy. However, with two versions, additional storage is now needed. If it is not provided, nearly every second transaction will abort, as the cache line containing the top of the stack is accessed too often. Thus, either the L1 cache has to be extended to store multiple versions of the same memory location or the rest of the memory hierarchy needs to be extended to support multiple versions. In the course of the project, it turned out that there is little difference in implementation difficulty

between a transactional memory system supporting two speculative versions or n speculative versions with n being an arbitrary constant. Therefore, at this point one could implement a multiversioning-based approach like described in Chapter 6 just as well.

4.5.2. Transaction Length Synchronization

For the transactions' checksums to match at commit, they need to commit at exactly the same instruction. However, it is nontrivial to identify the matching instruction on the trailing core. One cannot use the PC, as the instruction might be in a loop. It is also not possible to use intuitive performance counters like *retired instructions*, as they are nondeterministic [65]. In the course of this project, it turned out that this nondeterminism even occurs on simple in-order CPUs. Thus, it is required to identify the instruction through a combination of multiple indicators like PC, register values and deterministic performance counters. However, not even this might work if the leading and trailing core are heterogeneous. For example, the leading core used predicated execution [37] it might commit at an instruction, which the trailing core does not reach. In addition, transaction length synchronization by matching commit instructions is only possible if the trailing core cannot run ahead at all. This means, it is also necessary to reliably detect whether the (in-order) trailing core is currently ahead of the (out-of-order) leading core to pause it when needed.

4.5.3. Guaranteeing Transaction Commits

For this approach to work reliably, it is required that a transaction on the trailing core succeeds if the corresponding transaction on the leading core has succeeded. If the redundant copies share the same memory, it is not even possible to repeat the section without transactions, as the leading core has already updated memory locations, which might be read in the section. Thus, it would be necessary to resynchronize the trailing core, which is slow and not fault tolerant. This issue is intensified, as it is not possible to distinguish whether the abort was a true capacity abort or is caused by an erroneous memory access.

Guaranteeing commits is difficult for homogeneous system, not to mention heterogeneous systems. A small difference in cache state, can lead to a cache line being evicted on the trailing core before it is evicted on the leading core. Most HTM systems abort if this cache line was in the read- or write-set. Thus, cache state has to be identical for both cores. This is already hard to achieve in a shared memory multi-core and gets even more difficult if mechanisms like perfect prefetching are added. On a heterogeneous system, care needs to be taken to ensure that the read- and write-set are identical on both cores. For example, an out-of-order core might add a speculative read, which the in-order core never executes, to its read-set.

4.6. Summary

This chapter presented the single-threaded fault tolerance variant. This approach protects a single-threaded application, while it is in user mode, utilizing two cores. The approach can be disabled and execution of multiple redundant applications is possible if more cores are available.

The transactional memory was extended in various ways to support fault tolerance. Instead of explicit transaction bound instructions, automatic bounds are used. Transactions start right away and commit once any limit like cache size or execution time is reached. Error detection can be implemented by either calculating checksums or comparing modified cache lines at commit. The leading core supports two checkpoints simultaneously. Conflict detection is disabled while in fault tolerant mode.

The performance of the system can be improved by accelerating the trailing core. The leading core's memory accesses can be forwarded to the trailing core in order to realize perfect prefetching. The same can be done with branch outcomes to realize perfect branch prediction. As both optimizations only affect prediction mechanisms, error detection accuracy is not affected.

It is best to employ this approach on a heterogeneous multi-core. The leading core should be sufficiently fast and a prefetcher and a branch predictor should be available to implement the mentioned optimizations. The memory hierarchy should include a shared cache to reduce memory load.

There are several challenges when implementing this approach in hardware. Providing support for two checkpoints requires fundamental changes to the implementation of most HTMs. Synchronizing transaction commits is difficult due to the nondeterministic performance counters and differences between the cores. It is hard to guarantee successful transaction commits on the trailing core even if the leading core has completed the transaction.

5

Single-Threaded Evaluation

This chapter contains an evaluation of the single-threaded fault tolerance approach. We performed the evaluation in the gem5 simulator with focus on performance and power consumption. To realize this, we enhanced the simulator and implemented microbenchmarks. We already released this evaluation in shortened form as [4].

Table of Contents

5.1	Methodology	37
5.1.1	Implementation	38
5.1.2	Stride Prefetcher	39
5.1.3	Limitations	40
5.1.4	Benchmarks	41
5.2	Power Efficiency Evaluation	43
5.3	Summary	47

5.1. Methodology

The following subsections describe the methodology, which was used to evaluate the single-threaded fault tolerance approach. The evaluation was performed using an implementation in the gem5 simulator [16]. The optimizations to accelerate the trailing core are available, as well as a stride prefetcher. We used custom microbenchmarks for a fast evaluation time.

	Cortex-A7	Cortex-A15
Frequency	500 - 1300 MHz	700 - 1900 MHz
Pipeline	in-order	out-of-order
Superscalar	partial	yes
Floating Point	yes	yes
L1 Instruction Cache	each 32 kB (2x assoc.)	each 32 kB (2x assoc.)
L1 Data Cache	each 32 kB (4x assoc.)	each 32 kB (4x assoc.)
L2 Unified Cache	shared 512 kB (8x assoc.)	shared 512 kB (8x assoc.)

Table 5.1.: Specifications of the Cortex-A7 and Cortex-A15

5.1.1. Implementation

The presented approach is modeled in the gem5 simulator [16]. The out-of-order and in-order cores are configured to match the ARM Cortex-A15 and ARM Cortex-A7, respectively. The configuration to achieve this is based on Butko et al. [18]. An extract of the configuration is shown in Table 5.1. The per core 32 kB L1 caches are split into data and instruction caches, while the per core 512 kB L2 caches are unified. Power consumption is approximated as the product of the simulated benchmark run-time and the average power consumption of an Exynos 5430 SoC. It is assumed that a lockstep system runs as fast as a corresponding single-core machine, but consumes twice the energy.

For our approach, both the fast Cortex-A15 as leading core and the power-efficient Cortex-A7 as trailing core are simulated simultaneously. This is a favorable combination, as the Cortex-A15 can accelerate the Cortex-A7 to ensure high performance, while overall power consumption is low because of the Cortex-A7's energy efficiency. To realize an execution, like the one shown in Figure 3.2, limits on the slack are put in place. The trailing core is prevented from committing more instructions than the leading core. Therefore, it cannot overtake the leading core. This ensures that the trailing core never needs to maintain multiple speculative transactions and that data always flows from the leading to the trailing core. A limit is put in place to prevent the leading core from running too far ahead. The leading core stalls, when it has committed 1,000 instructions more than the trailing core, or if it tries to evict a modified cache line that the trailing core has not written yet. These limitations are put in place to simulate the limited capacity of the HTM. They also provide an upper bound on the required queue sizes. To simulate the isolation of the transactional memory, the cores operate on two different memory regions, which are each mapped to a different memory controller.

The two optimizations perfect prefetching and branch outcome forwarding (see Section 4.3) are implemented. Whenever the leading core encounters a cache miss, a prefetch is directly enqueued in the trailing core. As the prefetch takes some time to execute and the leading core cannot run ahead if it would evict a modified cache

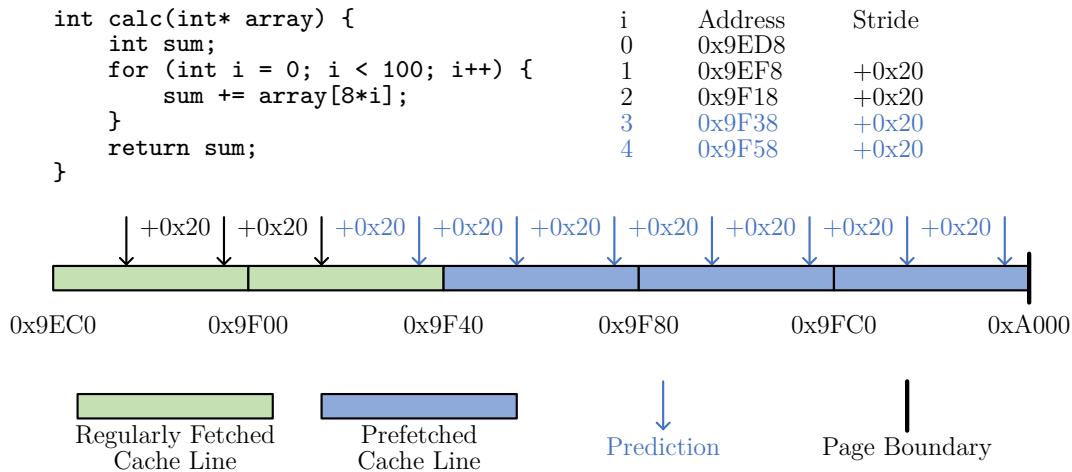


Figure 5.1.: The array load in the source code above can be optimized with a stride prefetcher. The first three loads are issued regularly and cause two cache lines to be fetched. After these loads, the stride prefetcher notices that the address appears to increase by 0x20. Therefore, it predicts which cache lines will be accessed next and prefetches them. After three cache lines, a page boundary is encountered, which prevents further prefetches.

line, which the trailing core has not written yet, no prefetch should occur too early. Branch outcomes are transferred using a queue (see Figure 4.2) and then matched by their PC. The cores in gem5 do not support any advanced mechanisms like predicated execution and they are based on the same implementation of the instruction set internally. Therefore, this simple matching mechanism is sufficient.

5.1.2. Stride Prefetcher

All configurations were evaluated with and without a hardware prefetcher. The Cortex-A7 utilizes early issue of memory operations [13] in all variants. A stride prefetcher [14], which tries to detect regular access patterns on a per-instruction basis, was used in the corresponding variants. If an instruction accesses memory locations with a constant distance, the prefetcher will predict the next addresses and preload them into the L1 cache. Detected streams are terminated at the page boundary, as the stride prefetcher works on physical addresses and two adjacent virtual pages are probably not mapped to two adjacent physical pages. Figure 5.1 shows an exemplary run of such a prefetcher. For this evaluation, we configured the prefetcher to observe 32 distinct instructions on a least recently used basis and prefetch up to 4 cache lines ahead of the last actual access.

5.1.3. Limitations

Because of simulation constraints and implementation complexity, this implementation has some limitations. Care was taken to minimize the impact of these limitations on the evaluation result. Each limitation is described below:

- As there is no baseline HTM system and there are no multi-threaded benchmarks in this evaluation, only the required features of the transactional memory were implemented. This also means that some overhead, which would be present in an HTM system, might be missing. However, this is primarily an implementation effort issue, as one would need to remove or disable those components, which generate the overhead, in the redundant implementation.
- The logic, which prevents the trailing core from overtaking the leading core, is implemented in the commit stage instead of the fetch stage of the trailing core. This has advantages for the implementation in the gem5 simulator. For example, logic to pause the commit stage is already available, while it has to be written from scratch for the fetch stage. Additionally, counting instructions is easier in the commit stage than in the fetch stage, as instructions might be speculative in the fetch stage. However, this shortcut causes issues with the forwarding logic. As branch prediction takes place in the fetch stage (at least in the gem5 implementation), the trailing core can exceed the branch outcome queue. Thus, it performs one invalid branch prediction. However, this only results in a performance penalty. As the correct implementation would also wait, this should not affect overall performance in a significant way.
- The power estimation only considers dynamic power consumption. Thus, the actual power consumption of all variants would be higher. As the Cortex-A15 is larger, its static power consumption is higher than the Cortex-A7's. Hence, the dual Cortex-A15 configuration would experience the largest increase and the dual Cortex-A7 configuration the smallest increase. The combined configuration would be in between. Therefore, the difference between measurements increases, but their order should not change.
- As the HTM system and checksums are not fully implemented, there is no error detection. Thus, rollback is also not simulated. However, the primary focus of the performance and power consumption evaluations is the error-free case. Those missing components should mostly be off the critical path in the error-free case and not influence the results. As the output of the benchmarks is correct and the statistics, which one would expect to match, are identical, it is very likely that the execution is properly duplicated.

5.1.4. Benchmarks

To keep evaluation times reasonable, we used a set of 6 microbenchmarks, which were chosen from well-known algorithms to represent a selection in memory access patterns, to evaluate the approach. This is necessary, as each benchmark has to be evaluated for every frequency and multiple configurations with a simulator, which incurs a high slowdown. However, we consider these benchmarks more appropriate for the intended use case of embedded systems than long running benchmarks. Our approach itself does not impose any restrictions regarding maximum runtime.

breadth-first

First, the *breadth-first* benchmark generates a random tree. A fixed seed is used for the random number generator. The tree is filled by inserting nodes (100,000 in this evaluation). The parent of each node is picked at random from all nodes, which were already inserted. This way it is ensured that all nodes are reachable. The nodes are linked by *child* and *next* pointers (in contrast to an ordered arrangement like in heaps). Thus, their memory arrangement is chaotic.

For the actual evaluation, the size of the tree is determined. The process starts at the root node. All children are counted and added to a queue. This process is then repeated for all nodes in the queue until the queue is empty.

heapsort

Initially, the *heapsort* benchmark generates a fixed size array (1,000,000 in this evaluation) consisting of random positive integers. A fixed seed is used for the random number generator.

The array is then sorted using heapsort [67]. First, a heap is generated in-place from the array. The heap is a binary tree, which is stored sequentially in memory. Thus, no pointers are needed to find the children or parents. During generation, it is ensured that the parent is always bigger than its children.

In a second step, the heap is converted to a sorted array. This is achieved by replacing each element with the root node, starting from the end of the array. After each swap, the heap is repaired to ensure its order. When the loop finishes, the result is a sorted array.

matrixmul

First, the *matrixmul* benchmark generates two matrices (8192x16 and 16x8192 in this evaluation) with random entries ranging from 0 to 9. A fixed seed is used for the random number generator.

The two matrices are then multiplied by strictly following the definition. For every entry in the result matrix, the corresponding row in the left matrix and the corresponding column in the right matrix are iterated simultaneously. The entries of the row and column are multiplied and summed up. The final sum is then put in the result matrix.

quicksort

Initially, the *quicksort* benchmark generates a fixed size array (1,000,000 in this evaluation) consisting of random positive integers. A fixed seed is used for the random number generator.

The array is then sorted using quicksort [31]. First, the last element of the array is selected as pivot element. All elements in the array are compared to this pivot element and the array is split into two subarrays depending on the outcome. To avoid the allocation of additional memory the Hoare partition scheme [31] is used. That is, an element at the beginning, which is larger than the pivot, and an element at the end, which is smaller than the pivot, are selected. Those two elements are then swapped. The partitioning concludes when the index starting at the beginning and the end cross. Finally, the pivot is placed between those two subarrays. The algorithm is repeated recursively on the subarrays until they are at most one element in length. This results in a sorted array.

red-black tree

The *red-black tree* benchmark generates a red-black tree [23] consisting of a number (100,000 in this evaluation) of random positive integers as nodes. A fixed seed is used for the random number generator. Every node consists of pointers to its parent and up to two children. Additionally, every node is either colored red or black.

A valid coloring has to follow two rules. First, a red node may not have any red children. Second, all paths from every node to the leaves contain the same number of black nodes.

The insertion algorithm is complex, as it consists of multiple cases involving recoloring and rotation steps. A detailed description can be found in [20, pp. 308-338]. For this evaluation, it is important to note that the branches and memory accesses are difficult to predict due to the pointer chasing and large amount of randomness involved.

shuffle

The *shuffle* benchmark generates an array containing a range of integers (0 to 99,999 in this evaluation) in random order. A fixed seed is used for the random number generator. The benchmark creates the array sequentially using a variant of the Fisher-Yates shuffle [21, pp. 26-27]. For each element, a random number between 0 and the

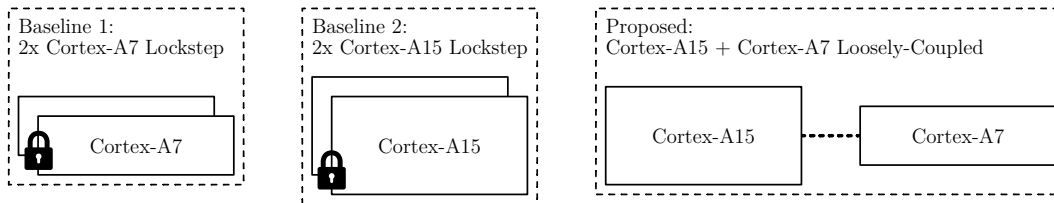


Figure 5.2.: Three configurations were evaluated. A lockstep system consisting of two Cortex-A7 and a lockstep system consisting of two Cortex-A15 form the baseline. For our approach a combination consisting of one Cortex-A15 and one Cortex-A7 are evaluated.

current insertion index is picked. The element at that index is moved to the end and then replaced by the insertion index. The resulting array contains every number exactly once and their probability distribution is uniform.

5.2. Power Efficiency Evaluation

The evaluation below focuses on throughput and power consumption. It consists of three different variants (see Figure 5.2), which are compared to each other. A lockstep system consisting of two Cortex-A7 and a lockstep system consisting of two Cortex-A15 form the baseline. The proposed single-threaded fault tolerance approach is implemented by a combination of a Cortex-A15 and a Cortex-A7. The Cortex-A7 is a power-efficient in-order CPU. Therefore, the corresponding baseline also shows low power consumption in the evaluation, but the throughput is also low. On the other hand, the Cortex-A15 is a high-performance out-of-order CPU. Thus, the corresponding baseline shows maximum throughput, but at a high power consumption.

Figure 5.3 shows the throughput and power consumption for all microbenchmarks and variants without the stride prefetcher. The two optimizations perfect prefetching and branch outcome forwarding are enabled. At first, only a small increase in voltage per 100 MHz step is required to ensure that the core to runs stable. Thus, for the lockstep systems, a large increase in throughput can be achieved at low frequencies by a small increase in power consumption. Note that the frequency itself has only minor influence on the results, as power consumption is measured per benchmark run and not per time unit. When the cores approach their maximum frequency, the required increase in voltage raises. At the same time, the achieved acceleration decreases, as the memory clock frequency remains constant. Thus, for high frequencies only a small increase in throughput can be achieved by a large increase in power consumption. The effect is more pronounced on the Cortex-A15, as it performs more instructions per cycle on average and its maximum frequency is higher.

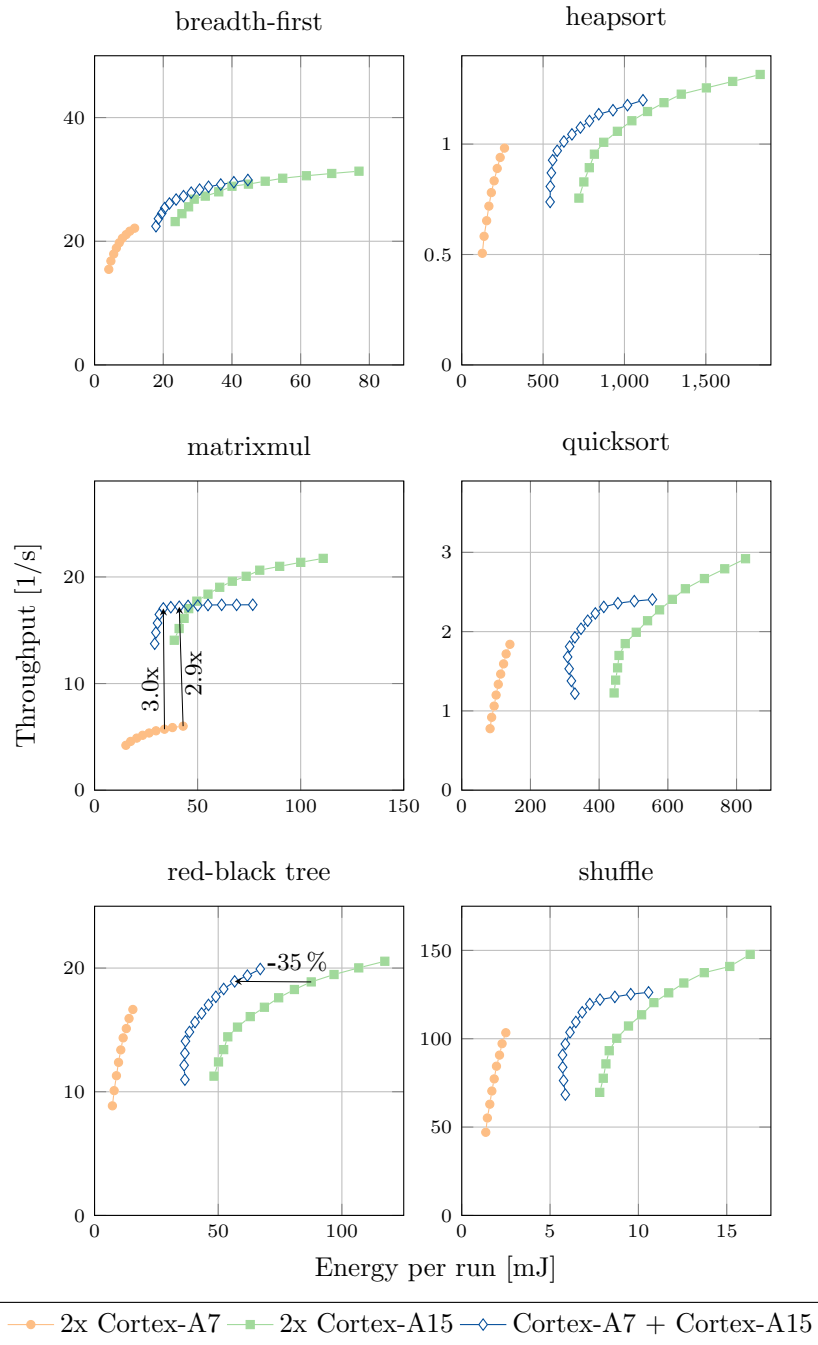


Figure 5.3.: The microbenchmarks' throughput is shown on the y-axis and the corresponding energy consumption per run on the x-axis. The microbenchmarks were executed on a lockstep system consisting of two Cortex-A7, another lockstep system consisting of two Cortex-A15 and our approach, using a Cortex-A15 as leading core and a Cortex-A7 as trailing core. The clock frequency of the cores was varied in their frequency ranges (Cortex-A7: 500-1300 MHz, Cortex-A15: 700-1900 MHz) in 100 MHz steps for the lockstep systems. For our approach, the trailing core's frequency was fixed at 1300 MHz, while the leading core's frequency was varied from 700 MHz to 1900 MHz. The stride prefetcher is disabled.

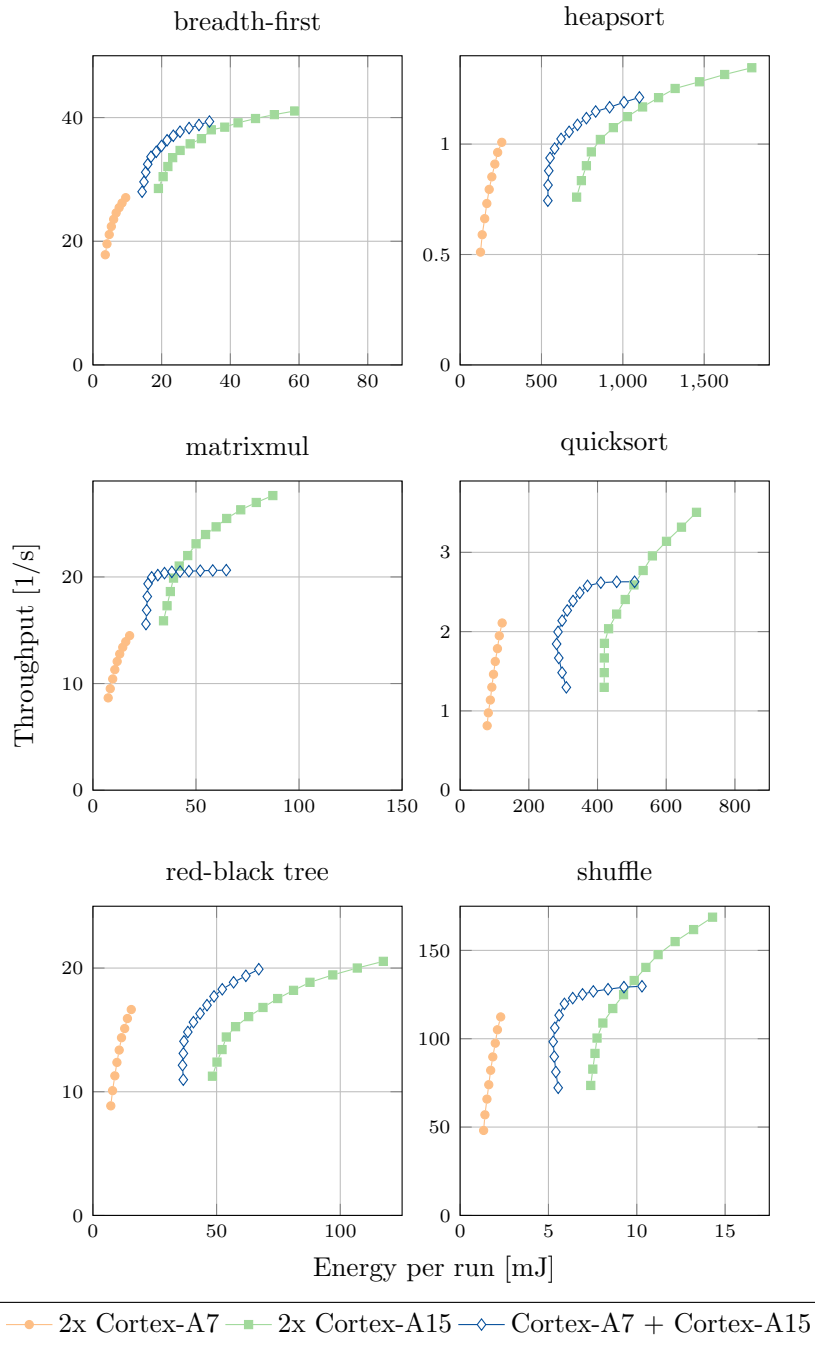


Figure 5.4.: The microbenchmarks' throughput is shown on the y-axis and the corresponding energy consumption per run on the x-axis. The microbenchmarks were executed on a lockstep system consisting of two Cortex-A7, another lockstep system consisting of two Cortex-A15 and our approach, using a Cortex-A15 as leading core and a Cortex-A7 as trailing core. The clock frequency of the cores was varied in their frequency ranges (Cortex-A7: 500-1300 MHz, Cortex-A15: 700-1900 MHz) in 100 MHz steps for the lockstep systems. For our approach, the trailing core's frequency was fixed at 1300 MHz, while the leading core's frequency was varied from 700 MHz to 1900 MHz. The stride prefetcher is enabled.

Our approach shows a different pattern. For the frequency range, in which the out-of-order core's performance does not exceed the in-order core's performance at maximum frequency, the leading core slows down the entire system. Increasing the leading core's frequency, can reduce total power consumption (e. g. in *quicksort* or *shuffle*), as the task finishes quicker, thus reducing the time the trailing core is running at maximum voltage. After the leading core's performance exceeds the trailing core's, there is a phase in which the trailing core can be accelerated to the leading core's level by forwarding the memory accesses. This area is the most interesting as it offers higher performance than the in-order core alone, at a lower power consumption than a lockstep system of two out-of-order cores. If the leading core's frequency is increased further, eventually a point will be reached, at which every memory access is prefetched. The graph asymptotically approaches this performance level. As the leading core's power consumption still raises, the combination will eventually consume more energy than a lockstep system consisting of two out-of-order cores would at the same performance level (apparent in *matrixmul*). Thus, further increasing the frequency of the leading core should be avoided.

The *matrixmul* benchmark shows the greatest increase in throughput over the lockstep system of two in-order cores. At best, our approach runs three times as fast at the same power consumption. It also beats the maximum throughput of the lockstep system of two in-order cores by a factor of 2.9 at the same power consumption. The *red-black tree* shows the largest reduction in power consumption over the lockstep system of two out-of-order cores. Our approach consumes 35 % less power at the same throughput.

Figure 5.4 shows the throughput and power consumption for all microbenchmarks and variants with optimizations and the stride prefetcher enabled. The effectiveness of the stride prefetcher varies depending on the benchmark's memory access pattern. For benchmarks with regular access pattern like *matrixmul* most cache misses can be eliminated by the prefetcher. This leads to a huge performance increase during the initialization of the source matrices. This phase does not profit from an increase in clock frequency, as it is entirely limited by the memory accesses. Out-of-order execution does not help much as well, as the reorder buffer cannot hold enough loop iterations to reach the next cache line but one. The prefetcher on the other hand can fetch four cache lines ahead. If the prefetcher is enabled, all variants profit from a performance increase at all clock frequencies. The calculation phase still hits the same limit in the Cortex-A7, as the variant without the stride-prefetcher already prefetches all memory accesses if the Cortex-A15 is clocked high enough.

The benchmark *shuffle* accesses one of the locations for the swap operand at random. As consequence, the stride prefetcher is unable to predict this access. The other swap operand can be prefetched. As multiple values fit in a cache line, the number of cache misses caused by this access is already lower to begin with. Therefore, the performance increase for the lockstep systems is relatively small. Our approach reaches the Cortex-A7's peak performance even without the stride prefetcher. With

a stride prefetcher, however, it is possible, to clock the Cortex-A15 slower and thus decrease total power consumption.

Tree-based benchmarks like *breadth-first* or *red-black tree* show a very irregular memory access pattern. They do not benefit as much from a stride prefetcher, as it will rarely detect consistent strides when traversing the tree. Therefore, the performance is exactly the same for *red-black tree* as without stride prefetcher. However, the stride prefetcher can improve performance for the queue used in *breadth-first*, as it shows a regular access pattern. An overly aggressive prefetcher may reduce performance for such algorithms, as it evicts cache lines that will be reused for false prefetches. Our approach on the other hand can eliminate all cache misses even for such irregular patterns, as long as the leading core runs fast enough. The resulting speedup exceeds, what is achievable with a simple prefetcher.

Our approach can achieve higher speedups than the stride prefetcher alone for both sorting algorithms. However, the reasons differ. The benchmark *heapsort* shows an irregular access pattern, as the heap is tree-based. Thus, our approach can benefit from its superior prefetching performance, while enabling the stride prefetcher results only in minor performance improvements. The benchmark *quicksort* on the other hand shows a very regular access pattern, as it linearly accesses elements from both directions. However, *quicksort* uses data dependent comparisons as loop condition in the Hoare partition scheme. Regular branch predictors cannot predict those branches, as they are essentially random for random data. However, in our approach the trailing core can use the forwarded branch outcomes from the leading core to further increase performance. Combining our approach with the stride prefetcher increases the throughput even further.

5.3. Summary

We implemented the approach in the gem5 simulator. We evaluated it in regards to throughput and power consumption on a combination of a Cortex-A15 and a Cortex-A7. The corresponding lockstep systems act as baseline.

The evaluation shows that the Cortex-A7 is able to keep up with the Cortex-A15 at lower frequency. Therefore, the power consumption of our approach is significantly lower than a lockstep dual Cortex-A15 system if this is the required performance level. However, for all benchmarks, except the tree-based ones, the performance of the dual Cortex-A15 system will eventually exceed the throughput of our approach. At this level, classic lockstep is preferable, as our approach only wastes energy while waiting for the Cortex-A7.

In the best case, our approach offers three times the throughput of a lockstep system consisting of dual Cortex-A7 at the same power consumption. Alternatively, it can

consume up to 35 % less power than a dual Cortex-A15 system at the same throughput. For all applications, there is a range, in which our approach runs faster than a dual Cortex-A7, while also consuming less power than a dual Cortex-A15.

6

Multi-Threaded Fault Tolerance

This chapter describes the multi-threaded variant of the fault tolerance approach in general. When implementing multi-threaded fault tolerance, there are challenges like execution order, input synchronization and thread synchronization. In this variant, data multiversioning with loose coupling is used to resolve these challenges. This chapter explains the concept and operation of the proposed multiversioning system. The detection and handling of conflicts is described. The mechanisms that realize fault tolerance are also explained.

Table of Contents

6.1	Challenges in Multi-Threaded Fault Tolerance	50
6.1.1	Execution Order on Leading and Trailing	50
6.1.2	Input Synchronization	50
6.1.3	Thread Synchronization	52
6.2	Multiversioning	52
6.2.1	Concept	52
6.2.2	Operations	54
6.2.3	Version Metadata	55
6.2.4	Version Management	56
6.3	Transaction Conflicts	58
6.3.1	Fundamentals	58
6.3.2	Detection	58
6.3.3	False Sharing	59
6.4	Fault Tolerance	60
6.4.1	Association of Leading and Trailing Cores	61
6.4.2	Required Version Count	62

6.4.3	Comparison	62
6.4.4	Rollback	64
6.5	Summary	66

6.1. Challenges in Multi-Threaded Fault Tolerance

There are multiple challenges when implementing fault tolerance for systems that execute multi-threaded applications.

6.1.1. Execution Order on Leading and Trailing

One of the main sources of indeterminism in multi-threaded applications is the order in which the threads are executed. The main cause of these differences in execution order is the operating system scheduler. If more threads are ready to execute than there are cores on the system, the operating system has to decide, which thread to execute. These decisions are also influenced by other processes. Therefore, the thread executed next by the examined process might differ, as other processes change scheduler state.

However, differences in execution order are possible even without an operating system. For example, small differences in initial cache or branch predictor state might result in different threads entering a critical section first. This causes an even larger divergence, which could influence another synchronization construct. Thus, the difference will most likely build up over time.

These differences in execution order can result in different memory states. This is highly problematic for fault tolerance approaches utilizing redundancy, as the redundant copies then do not match. Figure 6.1 shows an example of such a situation. If the rollback results in the same state again, the system might even get stuck. Our approach supports multiversioning, which stores multiple versions of each memory word. Multiversioning solves the issue of different execution orders, as can be seen in Figure 6.2.

6.1.2. Input Synchronization

If an application, which is executed redundantly, performs an operating system call, the output of this call must be written to both copies. The same applies to values, which were read from processor pins or memory-mapped devices by bare metal applications. For single-threaded applications, this can be realized easily: Both copies have to reach the system call or read operation. During the system call, no other code is accessing the application memory. Thus, the result can simply be copied from one to the other.

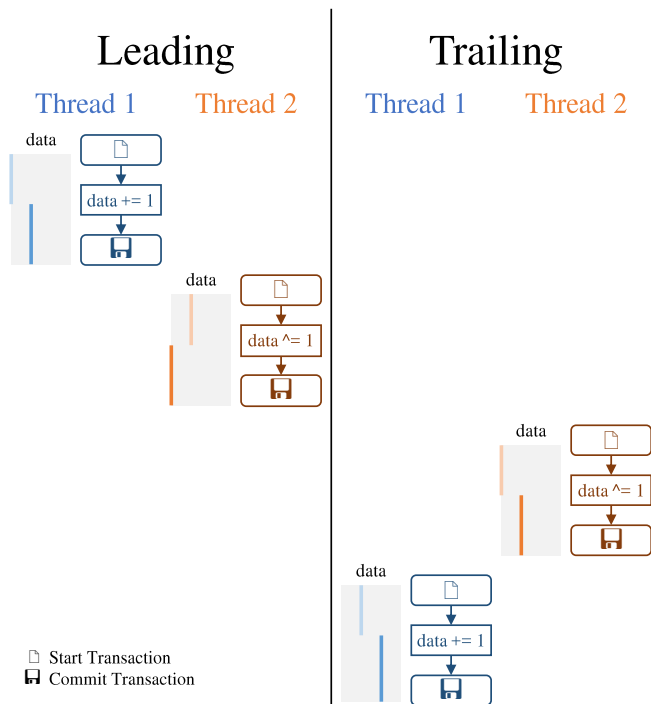


Figure 6.1.: This incorrect execution can occur if the order of the transactions is not preserved between leading and trailing. Thread 1 wants to execute a transaction that increments data by 1. Thread 2 wants to execute a transaction that xors data by 1. If thread 1 is executed first (leading case in this figure), the final result is 0. On the other hand if thread 2 is executed first (trailing case in this figure), the final result is 2. This causes a mismatch in checksums and thus a rollback.

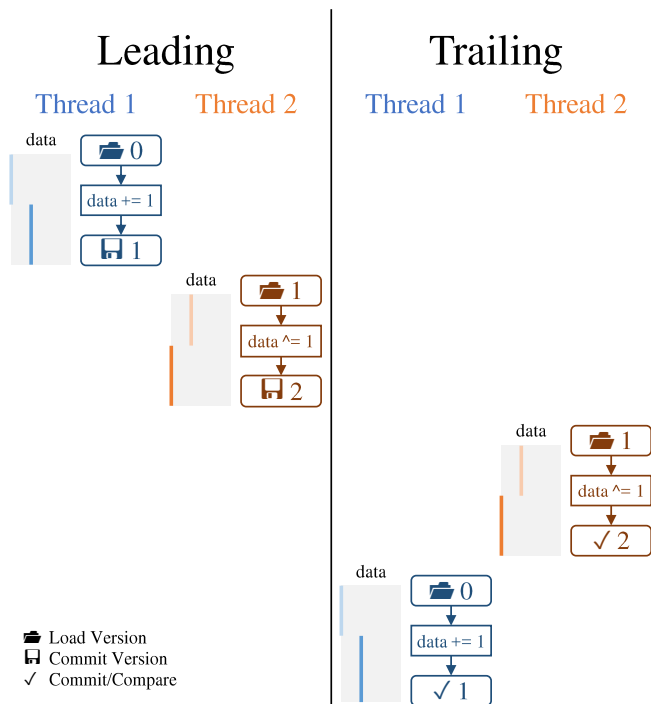


Figure 6.2.: The same application as in Figure 6.1 is shown here, but this time executed with multiversioning. From the perspective of the software, the execution on the leading core behaves the same. The result after the first transaction is stored in version 1 and the result after the second transaction in version 2. Even if the thread 2 is executed first in the trailing execution, it still loads version 1 as base. This is possible, as the version is still retained from the leading execution. Thus, the final result is also 0. Version 1 is validated later and the speculative resources are dropped, as version 2 is already available.

However, in a multi-threaded application another thread might be running and accessing the affected memory region. Therefore, it is necessary to realize the input duplication in a synchronized manner. Redundant processes must not store temporary values, which are not reflected in the other, in user space, as these might get accessed, too. In our approach, the input synchronization is handled by the multiversioning system. Transactions ensure that no temporary values are visible to other threads.

6.1.3. Thread Synchronization

Multi-threaded shared memory applications require synchronization between their threads. For non-redundant applications these synchronization primitives are usually implemented using atomic operations. However, many redundant approaches cannot guarantee the correct functionality of these atomic operations. For example, an atomic increment, which is executed on multiple threads at the same time, might return different values depending on the mapping between threads and cores. Additionally, other threads might not even see the effect of the atomic operation if an inappropriate isolation is used to contain faults. If an approach uses transactional memory, its synchronization capability is often lost, as conflict detection is disabled.

However, in our approach, transactions for synchronization are supported by the multiversioning system even in redundant mode. In the error-free case, both the leading and the trailing transaction always result in the same values. Classic atomic operations can be emulated using transactions (see Section 8.4.2).

6.2. Multiversioning

Multiversioning solves the challenges, which are mentioned above. In the following, we describe multiversioning in general. A concrete implementation on an FPGA follows in Chapter 7. This section starts by explaining the concept behind multiversioning. Then we describe the modes the system can be in and the possible operations. The required metadata is presented next. Finally, we detail the management of the version numbers.

6.2.1. Concept

The basic concept of multiversioning is that every memory location has multiple versions of its data. These versions have a defined order. When a core accesses the memory, it reads the last version, which is before a transaction-dependent upper bound. For a non-redundant or leading transaction, this bound is the maximum

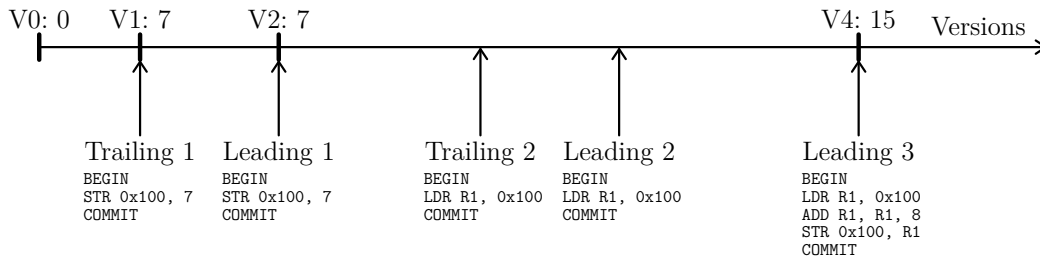


Figure 6.3.: Multiversioning can store different values for the same address. Bold vertical lines represent the different versions for the memory location 0x100. The labels above them indicate the version and the stored value. Note that the x-axis is not a time axis and all these versions exist at the same point in time (after the commit of the third leading transaction). There is a total of 4 different versions of the memory location 0x100. The version V0 contains the initial value, with which the memory was initialized at program start. The versions V1 and V2 both contain the value 7, which was written by the first transaction. There are separate versions for the leading and trailing transaction. The second transaction only reads the memory location (resulting in the value 7). Therefore, no separate versions were created. The third transaction was only executed by the leading core. It is still waiting for its execution by the trailing core. It adds 8 to the value, which results in another version V4 containing 15. Once the corresponding trailing transaction starts, it will be ordered between the second and third leading transactions. It will also add 8 to the value, resulting in version V3 (not shown). However, as the leading transaction has already committed, its result will not change. Thus, both transactions produce matching results.

possible value. However, for a trailing transaction, the bound is dependent on the corresponding leading transaction to avoid reading the leading transaction's modifications to main memory. Transactions write to a version, which is unique to this transaction. This version is created by the first write in that transaction.

Figure 6.3 shows an example of multiple transactions accessing the same memory location. The accessed version depends on the transaction. For non-redundant transactions or transactions on the leading thread, this version number is incremented for each transaction. Therefore, they reflect the serialization order of the transactions. If one looks at the most recent version, this view of memory matches how a memory, which only supports a single version, would behave.

Trailing transactions are always ordered right before the corresponding leading transaction. Therefore, they see the memory exactly as the leading transaction has, as they see all changes before, but not the changes made by the corresponding leading transaction or any later transactions. The changes made by the trailing transaction are invisible to all other transactions, as the leading transaction has also written the same memory locations and is ordered right after the trailing transaction.

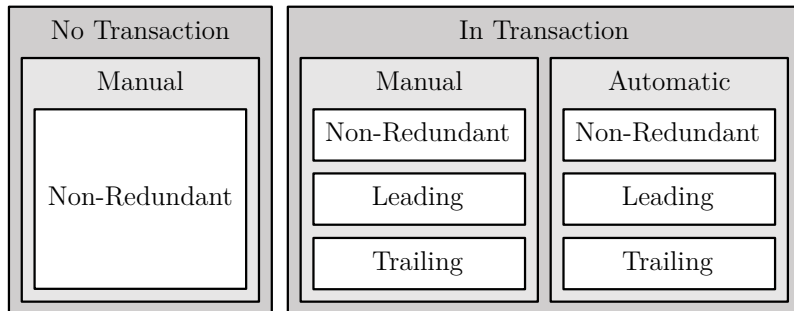


Figure 6.4.: Each core can be in one of several modes. There can be either an active transaction or not. Outside of transactions, redundant execution and automatic transaction launches are not possible. If there is a transaction, its bounds can be either manual or automatic. Independently, redundancy can be enabled or not. If redundancy is enabled, the core can be either a leading or a trailing core.

6.2.2. Operations

A core in a multiversioning system can be in multiple modes, which are shown in Figure 6.4. A multiversioning system can run without active transactions. In this case, only a single version is allowed. Fault tolerance is not possible in this mode. This mode is necessary for I/O operations, which cannot be executed in transactions, as most hardware components do not support rollback. Therefore, to perform an I/O operation, the thread has to leave transaction mode. Then the leading core performs the I/O operation alone. Finally, a new transaction is started and redundancy is re-enabled. The multiversioning system ensures input duplication.

The core has to run in transaction mode to support fault tolerance. In this case, each core is either a leading or a trailing core. Transactions without fault tolerance are also possible. In all cases, it is necessary to decide whether the CPU or the programmer define the transaction bounds. Note that the first transaction of a chain of automatic transactions is started explicitly. Only the following bounds are defined automatically.

A control register is used to configure the mode. This control register contains bits, which define the fault tolerance mode and bounds mode. When writing the control register, it is also necessary to define whether the change should commit or abort the current transaction. By using different bit combinations, the usual transaction control actions begin and commit can be realized. However, an additional action, called transaction fence, exists. In this action, a core, which is already in a transaction, starts a new transaction by committing the old transaction. Therefore, it behaves as if there was a commit followed by a begin executed in an atomic manner. The idea is that this action behaves similar to a classic shared memory fence. All memory accesses before the fence become visible to other cores, while no later memory access can pass the fence.

There are some limitations, which actions the trailing core can perform by writing to this control register. It is expected that the trailing transaction behaves the same as the leading transaction. As the leading transaction must have committed, it is always assumed that an error has occurred if the trailing core tries to abort a transaction. The trailing core cannot leave trailing mode, either because a leading core would never leave trailing mode, as it has never entered it. Attempts to leave leading mode are also ignored, as the trailing core has never entered leading mode. However, they are not assumed to be an error, as the leading core might be leaving leading mode to perform a system call or I/O operation.

Due to these limitations, the trailing core cannot leave redundant mode by executing an instruction. Therefore, it has to leave it by an external influence like an interrupt. When an interrupt is handled, the core automatically enters non-redundant mode, as it has to interact with the interrupt controller, which does not support rollback. If the system is supposed to execute redundant as well as non-redundant applications, the operating system has to setup timer interrupts before entering trailing mode. Those timer interrupts can then be used to leave trailing mode and to perform a context switch. When control switches back to the redundant process, the operating system is responsible to restore trailing mode. The operating system should also take care to avoid situations, where only leading cores or only trailing cores are available for a certain process, as this prevents it from making progress.

6.2.3. Version Metadata

In addition to the actual data, metadata has to be stored for each version:

Valid: This bit indicates whether the given version is valid or unused. You can also calculate this bit based on whether all other metadata is valid. However, the *valid* bit is needed for all memory operations. To achieve a low cache hit latency, it is preferable to have this information available without additional calculations.

Clean: This bit indicates whether the version contains changes, which have not been committed, yet. It is useful for conflict detection.

Timestamp: This integer ensures that the version are arranged in the serialization order of the transactions. This value must not overflow, as this would result in transactions reading the wrong value. A reset on overflow is expensive to realize, as it requires iterating over the whole memory to reset all versions at once. Regularly, one uses a global counter, which stores the current timestamp, to set this value. One can reduce the required width by only incrementing the current timestamp at commit instead of every cycle.

Leading: This bit indicates whether the version was created by a leading or a trailing transaction. Non-redundant transactions behave as if they were leading transactions. It is needed to differentiate the versions created by the leading and trailing

thread, as they have the same *timestamp*. It also prevents the trailing transaction from reading the data, which was written by the leading transaction.

Core id: This integer indicates the core, which has written this version. It is needed to detect write-after-write conflicts and to resolve commits, which occur in the same cycle. One can omit this value by integrating it in the low bits of the *timestamp* for uncommitted writes. If this is done, no two transactions may commit in the same cycle, as the trailing cores could not differentiate their versions. However, this value can also be advantageous for debugging, as it can be used to determine, which core has last written a cache line.

The size and storage location of these attributes is implementation dependent. Details follow in Chapter 7.

The values *timestamp*, *leading* and *core id* are used to order the versions. *Timestamp* is the most significant, as a version with a higher *timestamp* is always later in the serialization order. *Leading* comes next to ensure that a trailing transaction does not read the data of the corresponding trailing transaction. *Core id* is the least significant, as it is only relevant if two transactions commit in the same cycle. It has to be less significant than the *leading* bit, as the trailing transactions should not read any data of either leading transaction.

6.2.4. Version Management

To ensure correct execution careful management of the version numbers is needed. An example is shown in Figure 6.5. The behavior changes depending on whether the core is in leading or in trailing mode. *Valid* is always set, when the version is written, except on aborts. *Clean* is set on write and cleared on commit. The *core id* is set to the id of the executing core. *Leading* is set, when the executing core is not in trailing mode.

A transaction in leading mode on a multiversioning system aims to behave like a regular transaction for synchronization. Therefore, the four operations, read, write, commit and conflict detection, need to be considered. The processor always uses the most recent version written by a leading transaction for reads. In the actual implementation, one can ignore the *leading* bit completely, as every trailing version has to be shadowed by a leading version. Writes use the maximum possible timestamp. If another core accesses a cache line with this *timestamp* and the *core id* does not match, it knows that a conflict has occurred. As writes use the maximum timestamp, they can always be detected. On commit, the *timestamp* is changed to the current one to ensure the final timestamp can be used for the next transaction again. Note that this *timestamp* is not known while a leading transaction is running, but only at its commit.

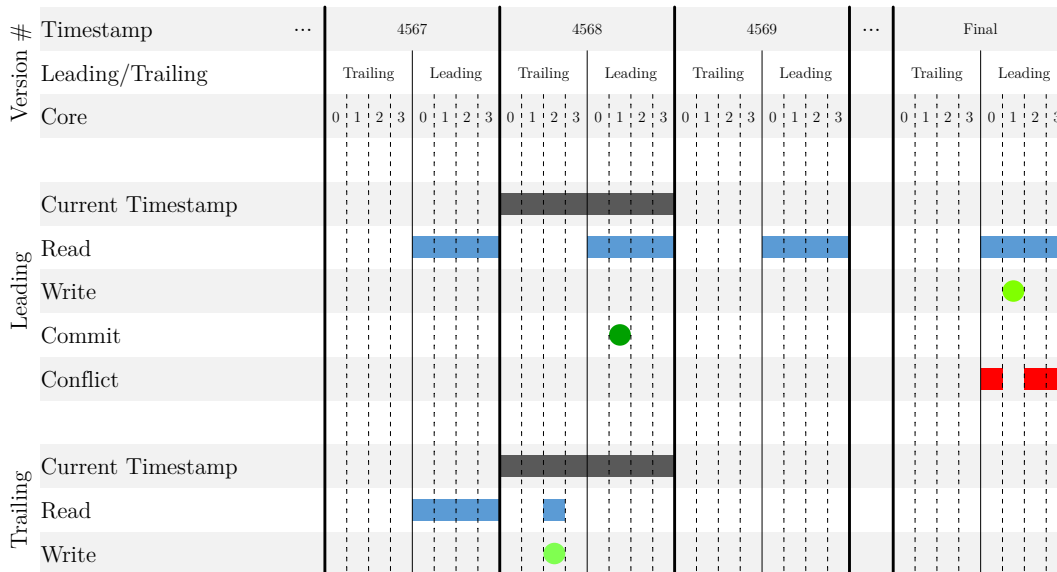


Figure 6.5.: An example how the version numbering works is shown above. The columns (except for the leftmost labels) represent different versions, while the rows (except for the three top labels) represent different operations. The affected versions regarded by the operation are marked with boxes and circles. Each version is uniquely identified by a combination of its *timestamp*, the *leading* bit and the *core id*. If a leading transaction attempts to read, it considers all versions that were written by any leading transaction. Those are marked by blue boxes. The actual instruction uses the value from the most recent version. It writes to the version (light green circle) indexed by the final timestamp (all bits are 1) and its *core id* (here 1). On commit, the *timestamp* is changed to the current timestamp (here 4568, dark gray box), resulting in the dark green circle. A conflict occurs if an accessed version has the final *timestamp*, is *leading* and the *core id* does not match. The versions, which cause this, are marked by red boxes. Trailing transactions adopt the *timestamp* from the corresponding leading transaction and write directly to this *timestamp*. They read either their version or the most recent previous leading version.

A transaction in trailing mode does not require all of the operations. It should not be involved in conflicts. Writes do not need to be persisted either, as they are shadowed by the corresponding leading writes, making the commit unnecessary. Therefore, only reads and writes are needed. The trailing transaction adopts the *timestamp* from the corresponding leading transaction. Consequently, the *timestamp* is known for the whole execution of the trailing transaction. It writes to the version with this *timestamp*. Reads use either this version or any previous leading version.

6.3. Transaction Conflicts

To use transactions for synchronization, it is necessary to detect and handle conflicts. This section starts by describing the kinds of conflicts, which can occur, and when they need to be handled. Then their detection is detailed. Finally, the problem of false sharing is described and remedies are pointed out.

6.3.1. Fundamentals

To support transactions for synchronization conflict detection is needed. While a transaction is running, read and write operations are performed. If two different transactions access the same memory location, a conflict occurs. For implementation reasons, most HTMs only track conflicts at cache line granularity. The most common way to resolve a conflict is by repeating one of the transactions. However, this is not necessary in all cases: If both transactions have only read a common memory location, the repeated execution would result in the same outcome. Therefore, it is not necessary to abort any transaction. If both transactions have only written a common memory location, the repeated execution would result in the same outcome. The final memory value would then be the one of the repeated transaction. Therefore, it would be possible to take this value directly without aborting the transaction. However, merging changed cache lines is difficult, so most HTM systems still handle it by aborting one of the transactions. If one transaction has read the common memory location, while the other one has written it, aborts are the only viable way to guarantee equivalent outputs to a sequential execution. Especially, if there are multiple conflicting memory locations, solutions, which try to preserve the initial execution become too complex very quickly.

The use of transactions for synchronization is only needed for leading cores. Trailing cores do not conflict with leading cores or other trailing cores, as the leading cores have already handled synchronization. Additionally, the trailing cores only replicate successful transactions, as aborted transactions do not influence system state. Therefore, a confirmation of aborted transactions is not required.

6.3.2. Detection

Detection of the conflicts is realized with the metadata. The *clean* bit and *core id* in the version metadata are used for conflict detection. If a leading core attempts to read or write a cache line, where the latest version is not committed and was written by another core, a conflict is detected.

Additionally, the *timestamp* of every accessed version is stored for read cache lines. At commit, the read-set is iterated and the latest version is compared to the stored version. If the *timestamps* do not match and the *core id* is different from the current

one, a conflict is detected, as the cache line was changed in the meantime. If the *core id* matches the current one, no conflict occurs, as this is a cache line, which was updated after it was read by the same transaction. It is also not possible for another update to have occurred, as the *clean* bit is not set, which causes every other core, which tries to write the cache line, to detect a conflict. Note that the validation of the read-set does not need to be atomic. The write-set is still write protected, as the *clean* bit is not set, yet. Therefore, the transaction can be ordered at the beginning of the commit even if the commit takes multiple cycles.

Altogether, the conflict combinations are detected in the following ways:

Read-after-read conflicts are not detected, as they do not require any special handling.

Write-after-read conflicts are detected by validating the read-set at the end of the transaction.

Read-after-write conflicts are detected by noticing the unset *clean* bit and differing cores on read.

Write-after-write conflicts are also detected by noticing the unset *clean* bit and differing cores on the second write.

6.3.3. False Sharing

False sharing is a common issue for most transactional memory systems and also regular shared memory systems. It occurs, when two cores access different words in the same cache line. Contrary to unsynchronized true sharing, where the same word is accessed, this does not affect the correctness of the program. However, the performance is affected negatively. In a regular shared memory system, cache line bouncing occurs. The cache line is first loaded exclusively by the first core. As the second core accesses it, the cache line needs to be transferred. If the core writes it, it has to be invalidated in the first cache. On the next access of the first core, another transfer is required.

In an HTM system, the effect is even worse. When the second core writes the cache line, either the transaction on the first or the second core is aborted, as a conflict is detected. It then has to be repeated. If the other core keeps accessing the cache line, transactions keep aborting, costing much run time. From a semantic standpoint, these aborts are not necessary, as the cores access different words. However, most HTM implementations require the abort, even if they can detect that the conflict is false, as merging the changes from a potentially unlimited number of transactions is difficult.

In a correctly written application, automatic transactions should never incur conflicts. To see this, imagine a hardware, where the size of every automatic transaction

is limited to a single instruction. If an application has conflicts in longer automatic transactions, it would incur race conditions, which potentially lead to wrong outputs, on such a hardware. If the application is ported from a shared memory implementation, the implementation does not incur conflicts in automatic transactions, as in essence a processor without transactions is such a single instruction transaction system. Therefore, we can assume that every conflict, which only involves automatic transactions, is due to false sharing. Depending on the kind of conflict, different optimizations can be implemented:

Read-after-read: If both transactions have only read the cache line, they can continue as normal. The cache line is then stored in shared state in both caches. This does not require any changes, as transactions do not incur conflicts in this case.

Read-after-write: If the first transaction has changed the cache line and the second attempts to read it, the first transaction can be committed early. This is allowed, as there is no guarantee, how long an automatic transaction runs. There is still some overhead, as the second core has to wait for the first commit to finish.

Write-after-write: If both transactions attempt to write the cache line, the first transaction can be committed early. There is still some overhead, as the second core can only execute the store after the commit of the first. However, the impact should be low in this case, as it can continue executing instructions as long as sufficient space is available in the load/store queue.

Write-after-read: If the first transaction has read the cache line and the second transaction updates it, the conflict can be suppressed. This is the only optimization, which can have negative side effects if the conflict was actually true. If the second transaction commits first, its version has a lower *timestamp*. Therefore, the trailing transaction will read its data. However, as the second transaction has changed the accessed word, it will read different data as the leading transaction, which causes a mismatch and rollback.

6.4. Fault Tolerance

This section describes the specifics of multi-threaded fault tolerance. The preceding sections have already described how fault tolerant execution is supported in the mechanisms of the multiversioning system. The following subsection explains the required ratio and the association of leading and trailing cores. Then the number of versions, which is required to realize fault tolerant with good performance, is illustrated. Afterwards, two methods to detect errors, checksums and comparison of written memory, are described. The final subsection explains how rollback in case of an error is realized.

6.4.1. Association of Leading and Trailing Cores

When running an application fault-tolerantly, every transaction, which was committed by a leading core, has to be confirmed by a trailing core. Therefore, a sufficient number of trailing cores is required and the work has to be distributed appropriately.

The exact ratio of leading to trailing cores depends on the platform. On a homogeneous system without optimizations, an equal number of leading and trailing cores is most likely the best choice, as execution on the trailing cores takes the same time as execution on the leading cores. However, on a heterogeneous system, it might be better to choose different core counts, when slower cores are used as either leading or trailing core. If optimizations like perfect prefetching or branch outcome forwarding (see Section 4.3) are used, it can be favorable to convert trailing cores to leading cores, as trailing transactions now take less time than the corresponding leading transactions. Additionally, it is not necessary to confirm aborted transactions, as they do not influence system state.

A dynamic ratio is also possible. However, changing the ratio with classic concepts is difficult. First, leaving trailing mode after it was entered is hard for a core, as it only executes instructions, which were also issued by a leading core. Second, the application has to support a dynamic number of threads, as it has to utilize a changing number of cores.

One can avoid these issues by using parallelization based on work items instead of threads. The application schedules all work items, which it wants to complete, on a hardware unit. For example, this can be realized by a configurable counter, which is automatically incremented in hardware. The hardware unit first checks whether a transaction awaits confirmation. If this is the case, this transaction run preferentially. Otherwise, the next work item is executed by starting a new leading transaction with a register set specific to the work item. Therefore, optimal utilization is guaranteed.

For both static and dynamic ratios, the system has to decide on which core a certain trailing transaction should run. In principle, any trailing core can be used independently of the leading core, as the multiversioning system guarantees input duplication. However, there are some criteria, which should be fulfilled, to ensure good performance. First, it is advantageous to execute trailing transactions in the same order in which the corresponding leading transactions have committed. This is beneficial, as versions can only be freed, once all earlier transactions have been confirmed, and freeing versions quickly ensures that no leading core needs to block because the version limit has been reached. Second, a trailing core should confirm multiple transactions of the same leading core if possible, as this benefits cache locality.

6.4.2. Required Version Count

A multiversioning system, which is intended for fault tolerance, has to support at least five versions per cache line: One safe version is required. This is the version the memory is rolled back to if an error occurs. All previous versions can be deleted. Additionally, at least one version is needed for the uncommitted leading transaction. Furthermore, the trailing core also needs a version for its uncommitted transaction. However, if only three versions are supported, performance will be poor. The confirmation by the trailing thread is not immediate. Therefore, the versions cannot be reused by the next leading transaction, which will likely access some of the same cache lines (e.g. the stack). This makes two additional versions necessary, which brings the total to five. Additional versions can be used to increase performance. However, this will also increase memory consumption and hardware complexity.

Once all previous transactions have been confirmed a leading version can be turned into the new safe version. Then the previous safe version can be freed. Trailing versions can be freed immediately after the corresponding trailing transaction has been completed, as they will never be read in another transaction.

Note that the trailing versions have to be reserved. Otherwise, the leading cores could consume all versions, leaving none for the trailing cores. This would prevent the confirmation of any more transactions. Therefore, the safe version cannot be advanced, which prevents any versions from being freed. Consequently, the system is in a deadlock state.

A possible optimization is to not write trailing versions to shared memory, but to use a core-local scratchpad memory instead. This can be done, as trailing versions are never accessed by any core except for the writer. This reduces the effective number of required trailing versions to one, as every trailing core only executes one transaction at a time.

6.4.3. Comparison

To detect errors, it is necessary to compare the results of the leading and the trailing transaction. This can be done using checksums. The checksum is updated with the result of every instruction. Memory accesses are also included in the checksum. Care needs to be taken to ensure the checksum is the same for both kinds of cores on heterogeneous multi-cores. When the leading core commits, the checksum can be transferred to the trailing core using the same mechanisms, which are used to transfer the registers or timestamps. After the trailing core has finished its own transaction, it compares the two checksums.

If they match, the leading transaction is considered confirmed. When the oldest leading transaction is confirmed, the safe version can be advanced. Resources, which

were allocated to store the old versions, can then be freed. If they do not match, rollback (see Section 6.4.4) is initiated.

Checksums suffer from the disadvantage of potential collisions, as the length of the checksum is limited and many states map to the same checksum. If a collision occurs, the error will not be detected. For this use case, a uniform distribution of the checksums is advantageous. If we assume that the effect of the error is unpredictable, the faulty execution's checksum is distributed uniformly with independent probabilities. Therefore, the probability, that a collision occurs is $\frac{1}{2^n}$ for a n bit checksum. As errors should already be rare, reasonable checksum sizes, like 32 bit, should be sufficient.

There are checksum algorithms like CRC (Cyclic Redundancy Check) [45], which can guarantee that every single-bit error and burst errors below a certain length are always detected. However, this does not apply to this approach, as a single changed bit can influence the outcome of multiple instructions in the same transaction. Therefore, multiple non-adjacent bits in the checksum's input change and the condition for the guarantee is not fulfilled. Overall, CRC is a bad choice because the burst error detection capability of CRC results in a non-uniform distribution. Finding a good checksum algorithm for our approach is hard. On the one hand, algorithms with non-uniform distributions suffer from many collisions. On the other hand, cryptographic hash functions have uniform distributions, but are too expensive to calculate for every instruction.

Alternatively, the written memory can be compared. This is easily possible on a multiversioning system, as the trailing core sees the corresponding leading version, when it writes its own version. It can then compare them directly. The outcomes of the comparison are handled the same as with checksum comparison.

This approach has multiple advantages: First, it is useful for debugging. While the checksum approach only reveals the transaction containing the mismatch, this approach specifies the transaction and the cache line. Therefore, issues with errors induced by false sharing are easier to find. Second, it offers better fault detection accuracy, as it does not suffer from the same collision problematic as checksums. Third, it is more immune against microarchitectural differences. Checksums will differ if there are different instruction outcomes (e.g. because of predicated execution on one of the cores) even if they do not influence the final output. However, those microarchitectural differences should never influence the data written to memory. Therefore, this approach is not affected.

However, this approach suffers from some disadvantages: First, it only monitors the memory content. Therefore, it is necessary to write all data, which could influence program execution, like registers or carry bits to memory. Additionally, missed writes cannot be detected easily, as the trailing core is not aware that it needs to check these memory locations. To remedy this, one has to either transfer the addresses

of all written cache lines to the trailing core or perform additional checking on the leading core. Both alternatives are expensive to implement.

6.4.4. Rollback

If an error is detected, rollback is required to recover from it. It is not sufficient to recover just the leading/trailing pair, which detected the error, as other cores might have already read the corrupted data, while the trailing core was still validating. Instead, every overlapping transaction has to roll back to its beginning. Figure 6.6 shows which transactions can be affected by an error. It is also necessary to ensure that all previous transactions are already validated, as they might imply an earlier rollback point.

A possible optimization is to roll back other transactions only to the point right before the commit of the transaction, which detected the error. However, this optimization does not increase performance in a meaningful way, as errors should be rare, but poses multiple challenges: If multiple faults occur at the same time, all those transactions have to roll back to their beginning, while all other transactions only roll back to the commit point. Therefore, handling for this has to be implemented in hardware. This is also not a rare case, which could be ignored, as the trailing transactions are not necessarily validated in order. For example, a fault might occur in an early transaction and then propagate to a later transaction. Possibly, this later transaction might be validated first and detects the mismatch. The first transaction then also detects the mismatch once it is finally validated and the complex variant of the rollback is necessary.

The optimization also requires conflict detection to work in the presence of bit-flips. However, this is not necessarily given. For example, optimizations to reduce false sharing (see Section 6.3.3) can impair conflict detection in presence of bit-flips. In the error-free case, two threads might access the same cache line, but different words. Therefore, no thread can affect the other. Though, a bit-flip could occur in the low bits of the address. Now, the data written by one thread might be read by the other, which results in a wrong result of the other thread, as it was not intended to read the data of the first thread. However, as it was assumed to be false sharing, the transaction is not aborted.

Redundant systems with uncoordinated checkpointing often suffer from an issue commonly referred as Domino Effect in literature [38, p. 209]. An error is detected on the first core and rollback is initiated. However, the core has already transmitted data to a second core between the checkpoint and detection of the error. Therefore, the second core requires rollback, too. If the checkpoint on the second core is older than the checkpoint on the first core, it might have sent data to the first core between them. This makes an additional rollback on the first core necessary, which could cause the cycle to repeat.

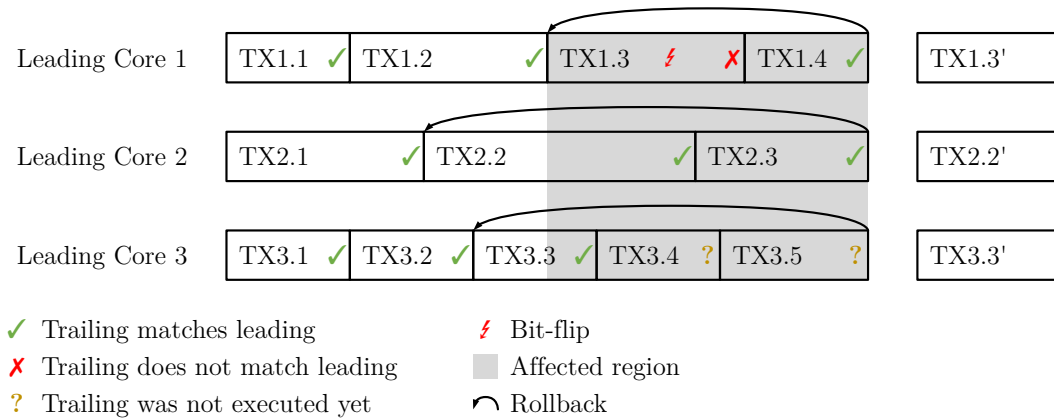


Figure 6.6.: At first, every transaction is confirmed without any errors. However, a bit-flip occurs in TX1.3, which makes global rollback necessary. At this point, every instruction executed in the gray area could possibly be corrupted. It does not matter that the result of the leading and trailing core match for TX1.4, as the bit-flip could have occurred in the leading core, which causes both leading and trailing core to read incorrect data in TX1.4. It is necessary to wait for the trailing cores to confirm TX2.2 and TX3.3, as an error in those transactions would cause a more comprehensive rollback. However, it does not matter that TX3.4 and TX3.5 were not confirmed by the trailing core yet. After the rollback point is determined and all active transactions are aborted, the execution is restarted at TX1.3', TX2.2' and TX3.3'.

Our approach uses uncoordinated checkpointing, but does not suffer from this issue. The HTM detects data transfers between uncommitted transactions and aborts one of them. Therefore, the rollback of the first core can cause rollbacks on other cores, as the error is detected after the commit of the leading transaction. For simplicity and as errors are rare these rollbacks are always performed in the algorithm described above. These rollbacks on the other cores cannot cause additional rollbacks on the first (or any other) core, as all transactions, which ran after the commit of a transaction that was rolled back, are rolled back themselves. This can be observed in Figure 6.6. All transactions, which committed after the start of the affected region, are rolled back. Therefore, all data transfers that take place in the affected region are already taken care of. Between the checkpoints and the start of the affected region there cannot be any data transfers as no transaction, which was rolled back, has committed yet and transfers originating from an uncommitted transaction would have been detected as conflict. Note that the argumentation still holds even if the system is optimized for false sharing, as conflicts, where the first core writes, are always detected and handled in some way.

6.5. Summary

There are multiple challenges, when implementing multi-threaded fault tolerance: Execution order on leading and trailing, input synchronization and thread synchronization

These challenges can be solved using multiversioning. A multiversioning system stores multiple versions for each memory word. These versions are used to ensure trailing transactions read the same data as the corresponding leading transaction, even when the cores are coupled loosely. Additionally, multiversioning can also be used for synchronization.

The proposed multiversioning system supports multiple modes: A core can run without redundancy, in leading or in trailing mode. Additionally, transaction bounds can be determined automatically or by the programmer. The system supports the common transactional memory operations. Furthermore, operations are available for mode changes.

Additional metadata is necessary to manage versions. This metadata is used to order the versions correctly and ensure proper conflict detection. Trailing transaction never encounter conflicts. We implemented optimizations to lessen the impact of false sharing.

For fault tolerant execution, redundant transactions are scheduled on trailing cores. A dynamic ratio of leading to trailing cores is also possible. At least five versions are required for fault tolerant execution. The result of the transactions is compared by either checksums or their memory writes. In case of a mismatch, the state of the whole multi-core is rolled back.

7

FPGA Implementation

This chapter describes the implementation of multiversioning on an FPGA. An FPGA-based implementation has to be more detailed than a simulation and is a good indication that the approach works properly. The MicroBlaze processor is a soft core developed by Xilinx and is used for our prototype. The L1 cache is replaced by a custom implementation supporting multiversioning with various additional data structures. The caches also handle conflict detection. The interaction with the MicroBlaze cores works over a memory-mapped interface. Additional devices like UART are available to run and debug the applications on the cores. Various optimization to improve performance can be enabled individually.

Table of Contents

7.1	Multiversioning Implementation	68
7.1.1	Memory Hierarchy	68
7.1.2	Concept	70
7.1.3	Data Structures	72
7.1.4	Hardware Overhead	75
7.1.5	Conflict Detection and Commit	76
7.2	MicroBlaze Interaction	76
7.2.1	Transaction Control	77
7.2.2	Transaction Bounds	78
7.2.3	Configuration	79
7.2.4	Register Backup	80
7.2.5	Transaction Abort	82
7.2.6	Trailing Execution	84
7.2.7	Performance Counters	84
7.3	Devices	87

7.3.1	Memory Controllers	87
7.3.2	UART	87
7.3.3	Debuggers	88
7.4	Optimizations	88
7.4.1	Validation of Automatic Transactions	88
7.4.2	Bloom Filter	89
7.4.3	Cache Line Compression	90
7.4.4	Fresh Fetch	90
7.4.5	Sticky Trailing Threads	91
7.4.6	Important Writes	91
7.5	Summary	92

7.1. Multiversioning Implementation

This section describes the implementation of our multiversioning approach in hardware. It starts with a description of the memory hierarchy. The general concept is explained next. Then, an overview of the used data structures is given. A description of the incurred hardware overhead follows. Finally, the process of conflict detection and commit is detailed.

7.1.1. Memory Hierarchy

We use MicroBlaze [73] cores as processors. These are closed-source soft cores developed by Xilinx specifically for the use on FPGAs. They offer support for hardware floating point and atomic instructions. A gcc compiler is available to compile C++ code to the MicroBlaze's RISC instruction set. We implement multiversioning by providing a custom cache. Modifications to the MicroBlaze cores are not required.

Figure 7.1 shows the memory hierarchy of our system. The MicroBlaze cores are connected to the cache with the three interfaces ILMB, DLMB and M_AXI_DP. LMB (Local Memory Bus) [71, 73] is an interface to connect processors to high-speed peripherals with low latency that is primarily used by Xilinx IP. AXI (Advanced eXtensible Interface) [41] is a more feature-rich, but also more complex interface by ARM that is used by a wide range of manufactures.

The port ILMB is used to fetch the instructions. As it is read-only, most signals remain unused. This port is connected to a custom instruction cache. This cache does not support multiversioning, but is coherent with the data caches to enable easy code updates, while the processor is running.

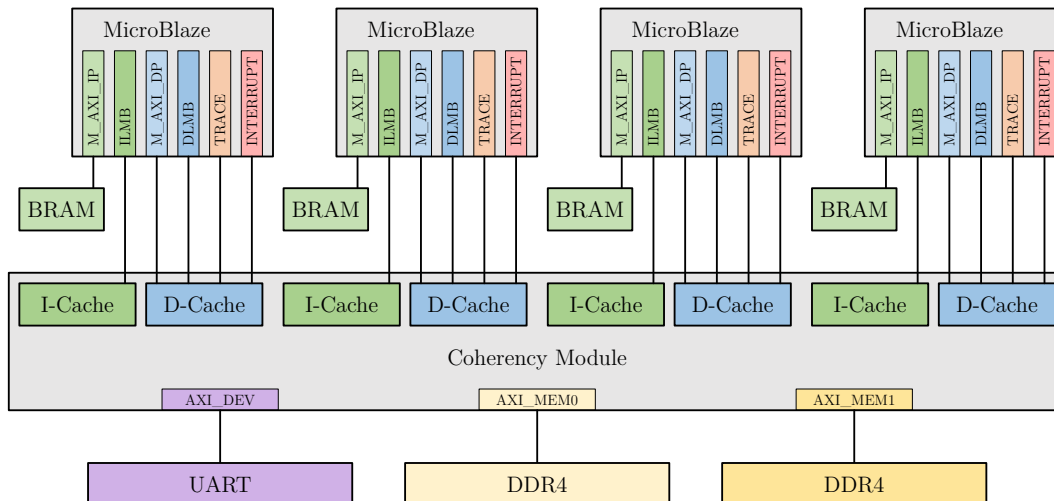


Figure 7.1.: The memory hierarchy of our system is shown above. MicroBlaze cores are used as processing units. Each core is connected to a block memory, a private instruction cache and a private data cache. All caches are contained in a coherency module. This module is connected to a UART controller and two DDR4 controllers. All components are also connected to various hardware debuggers (not shown).

The block memory connected to the `M_AXI_IP` port is not actually used. It is just there to ensure that the Xilinx tools work correctly. If the MicroBlaze cores are only connected to external memory, the memory controller does not work [62]. This is caused by some interference between the MicroBlaze processors intended for the application and the MicroBlaze processor responsible for calibrating the DDR4 memory. This is most likely a bug in Vivado, but there are no plans for it to be fixed. The `M_AXI_IP` port is convenient for this workaround, as it is not needed for multiversioning.

The ports `DLMB` and `M_AXI_DP` are connected to the data cache. As `LMB` does not support atomic instructions, the additional AXI port is needed. The `LMB` port has a lower latency than the AXI port. Therefore, it is preferred for regular memory accesses. The MicroBlaze processor does not map the ports by address range. Instead, the `LMB` port is always tried first and if it does not respond, the AXI port is tried next. We rely on the undefined behavior that atomics are signaled on the AXI port before it becomes valid. This way we can select the correct port to use for each memory operation without analyzing the instructions themselves. We do not expect this behavior to change, as internally both ports are connected to the same output signals of the `MEM` stage. Therefore, the AXI port changes, when the `LMB` port does.

In addition to the memory interfaces, the core's `TRACE` and `INTERRUPT` ports are also connected to the corresponding data cache. These ports are used to implement

transactional memory functionality. The TRACE port is required to create a copy of the register set for rollback. To signal conflicts to the core, an interrupt is raised.

All caches are part of the same module. Therefore, cache coherence can easily be realized, as every cache knows about the memory accesses of all cores. On a system with a higher frequency or more cores, this would not be viable. Such a system would most likely use directory-based cache coherency instead. As multiversioning metadata is realized as regular data (see Section 7.1.2), multiversioning is mostly independent of the cache coherence and only minor changes are needed if an alternate coherence approach is used.

The encapsulating module also handles all accesses to the underlying memory or devices. For this reason, two AXI ports AXI_MEM0 and AXI_MEM1 are provided. They are connected to two memory controllers by Xilinx [80], which address one DDR4 module each. Another AXI port AXI_DEV is used to communicate with devices. On our platform, only UART is available.

The lower half of the address space (0x00000000 - 0x7FFFFFFF) maps to the DDR4 memory. It is interleaved between the two memory controllers at cache line granularity. The upper half of the address space (0x80000000 - 0xFFFFFFFF) maps to devices. The memory-mapped interfaces provided by our implementation are also included in this range.

7.1.2. Concept

The version metadata matches the one described in Section 6.2.3. The total size of the metadata is 40 bits for each version. The size of the *core id* is dependent on the number of cores. For example, for 16 cores, 4 bits are needed. The *valid*, *clean* and *leading* fields consume one bit each. Therefore, for a 16 core machine, 33 bits are left for the timestamp. Assume a 16 core system that achieves a frequency of 100 MHz and an average transaction length of 10,000 cycles. As each (committed) transaction consumes one version, the timestamps would last for $2^{33} \cdot 10,000/16/100 \text{ MHz} = 53687 \text{ s} \approx 15 \text{ h}$. This is sufficient for an experimental system even without handling overflows. However, for a production system either a larger timestamp size or a mechanism to handle overflows is needed.

The space to store the version metadata and addresses of the versions is obtained by shifting every address left by one bit. Therefore, 64 additional bytes are available per 64 byte cache line. Note that in a production implementation version metadata would not be stored next to the cache line. Instead, all metadata, except for the safe version, would be stored next to each other in a compressed memory area. As most metadata is zero, the resulting memory overhead is much smaller than 2x.

Figure 7.2 shows how the space is used. Mode & index are only used by cache line compression (see Section 7.4.3). It consumes 8 bits for the mode and 6 bits for the

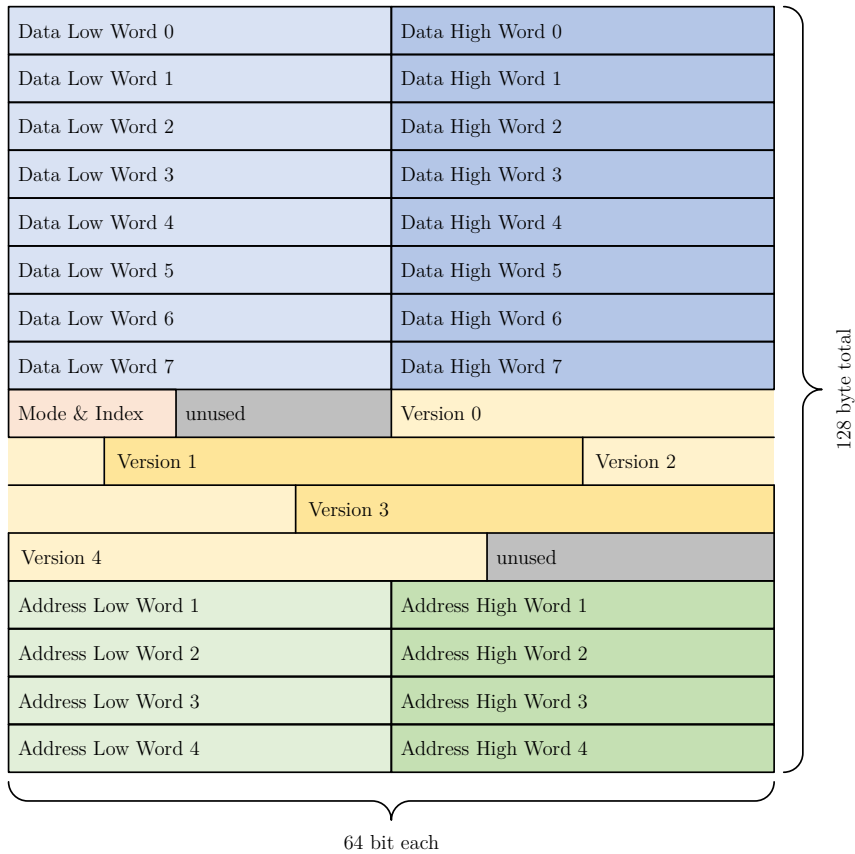


Figure 7.2.: Each cache line contains persistent (will be evicted to RAM) data. The metadata of the cache line itself (e.g. the tag) is not shown. Half of the cache line is used for the data (blue) of the safe version. This data is split into high and low words. The second half is used for versioning information. First, mode & index (red) are stored. Then, 5 version numbers (yellow) follow. Lastly, addresses (green) for all versions except for the safe version are retained. These are also split into high and low words.

index. Note that the address high word¹ is not actually used, as the MicroBlaze is a 32 bit system. However, it has to be present as cache line compression makes use of it.

The version metadata can be extended to 48 bit by making use of the adjacent unused space. However, a larger timestamp does not only require more space, but also larger comparators and adders.

The data of all versions except for the safe version are stored at a different location in memory. These redirected cache lines only contain data and no additional version information. Therefore, only 5 versions are available per cache line.

The MicroBlaze's memory operations are always 4 byte aligned on the bus. Misaligned and larger memory operations are split into multiple aligned operations. Short writes are realized with byte enable signals. This reduces the complexity of the cache implementation, as every data bit can be mapped to exactly one data line on the bus.

7.1.3. Data Structures

Several data structures are required to store the various required information and to access in an efficient manner. This list only contains the data structures used in the data cache or directly related to multiversioning. Scalar registers are not included, either. Note that memories are generally dual ported, as the underlying hardware structures on the FPGA are implemented this way.

Cache Tags

Type: Per Core Memory

Width: $([\text{tag size}] + 1) \cdot [\text{associativity}]$

Depth: $[\text{set count}]$

This memory stores the tags for all cache lines. All tags of one set can be accessed at once to determine the correct way in a single cycle. The dirty bit is also included in this memory, which accelerates writeback logic.

Cache Data

Type: Per Core Memory

Width: $[\text{cache line size}]/2$

Depth: $[\text{cache line count}] \cdot 2$

This memory stores all cache line data. It also includes multiversioning metadata. The memory is accessed twice per cycle, on the rising and on the falling edge. This reduces the width of the memory, which is favorable for an FPGA implementation.

¹In this dissertation, a word is a 32 bit value, in compliance with the MicroBlaze documentation.

Read-Set

Type: Per Core FIFO (First In - First Out) Queue

Width: $32 + \lceil \log_2([\text{version count}]) \rceil$

Depth: 4096

This FIFO queue stores the address and the version of cache lines, which were read in the current transaction. Cache lines, which were already written in the same transaction, are not stored. There can be duplicates if the same cache line is read multiple times. The depth is based on the underlying hardware structure. For most applications, a smaller depth should also be sufficient. The full address is only stored for debugging purposes.

Write-Set

Type: Per Core FIFO Queue

Width: $32 + \lceil \log_2([\text{version count}]) \rceil$

Depth: 4096

This FIFO queue stores the address and the version of cache lines, which were written in the current transaction. There are no duplicates. The depth is based on the underlying hardware structure. For most applications, a smaller depth should also be sufficient. The full address is only stored for debugging purposes.

Cleanup Queue

Type: Per Core FIFO Queue

Width: $32 + \lceil \log_2([\text{version count}]) \rceil$

Depth: 4096

This FIFO queue stores the address and the version of cache lines, which need to be cleaned up once they are confirmed. There are no duplicates. The depth is based on the underlying hardware structure. For most applications, a smaller depth should also be sufficient. The full address is only stored for debugging purposes.

Fallback Queue

Type: Per Core FIFO Queue

Width: 32

Depth: 4096

This FIFO queue stores free fallback cache lines. These cache lines are used to store the changed data if a new version is allocated.

Register Backup

Type: Per Core Memory

Width: $2 \cdot 32 \cdot 32$

Depth: 1

This memory stores the current register set. The state of the register set at the start of the transaction is also stored. As the whole state is copied in one cycle, when a new transaction starts, all registers need to be accessible at once. Register R0 is always zero, but is left in to simplify logic.

Read-Set Bloom Filter

Type: Per Core and Memory Controller Bloom Filter

Width: $1 \cdot 2$

Depth 16384

This Bloom filter contains all cache lines, which were read in the current transaction. Only a single hash function is used. Therefore, only one bit has to be read/written per cycle. As it takes a long time to clear the filter, two filters are available. When a transaction begins, the filters are switched and the reset of the old filter starts. It is only used if the corresponding optimization (see Section 7.4.2) is enabled.

Spinstats

Type: Per Core Counters

Width: 52

Depth: 18

This is a special structure, which stores all optimized performance counters in an efficient manner (see Section 7.2.7). Essentially, it consists of multiple counters, which are stored and incremented more efficiently than in a basic implementation.

Waiting Trailing Thread Queue

Type: Global FIFO Queue

Width: $32 \cdot 32 + 4 \cdot 32 + 15 + [\text{checksum size}] + \lceil \log_2([\text{core count}]) \rceil + [\text{timestamp size}] + [\text{important writes size}]$

Depth: 16

This global FIFO queue is used to transfer the state from the leading cores to the trailing core. It includes all registers, PC, MSR (Machine Status Register), abort count, number of executed memory operations, checksum, leading checksum modification count, ID of the leading core, leading timestamp and important writes (only if the optimization is enabled, see Section 7.4.6).

7.1.4. Hardware Overhead

One major factor that reduces the achievable frequency of the cache is the determination of the correct version to use. The version can potentially affect all output signals of the cache, as it can change the value returned to the CPU, cause a cache miss and determine the address, which needs to be fetched. Therefore, this determination is on the critical path inevitably. Identifying the correct version involves multiple comparisons (see Section 6.2.4). Firstly, every version has to be compared to the current timestamp of the active transaction. Next, the maximum version with a lower or equal timestamp has to be determined. The appropriate action (e.g. abort) is identified later by examining the *valid*, *clean*, *leading* and *core id* fields and does not influence most output signals.

To realize a low latency the comparisons are performed in parallel. One comparator is required per version to compare it to the timestamp of the active transaction. The safe version always has to be smaller than the timestamp of the active transaction. Therefore, this comparator can be omitted. Each version has to be compared to each other version to determine the maximum. However, half of the comparisons can be determined by inverting the result of the comparison with swapped operands. Therefore, the total number of additional comparators needed for 5 versions is $(5 - 1) + (5 \cdot 4/2) = 14$. This number is quite large, as normally only comparators equal to the associativity of the cache would be needed (not included in the previous calculation). The large number of parallel comparisons also greatly increases the width of the SRAM memory used for the cache, as all version metadata needs to be accessed.

With the current implementation, the required memory is doubled, as every cache line requires double the space, but contains the same amount of data (see Figure 7.2). Note again that a more sophisticated implementation can avoid this static overhead. In addition to the metadata, space has to be reserved for additional versions that cannot be stored in the same cache line. For simplicity, every core should have its own reserved area. As versions are cleaned up quickly, little memory is needed for this. Our implementation allocates 512 kB per core, which is sufficient in practice. If a large L3 cache is available, it could also be used to store the version data. This decreases the last level cache miss rate, as every version is needed again soon due to cleanup.

Even if the frequency can be kept constant, as it is limited by another component, multiversioning can negatively affect performance. If the accessed version is not the safe version, another cycle is required to access the data of that version. This might even cause another cache miss. If the capacity of the used SRAM memory is the limiting factor for the cache size, the number of stored cache lines has to be reduced. For this implementation, we assumed that the SRAM memory is not limiting and both a cache without and with multiversioning can store the same number of cache lines.

With current technologies, implementing multiversioning will certainly reduce cache performance. The full impact is not modeled in the FPGA implementation, as the impact of certain aspects (e.g. a changed SRAM ratio) is hard to determine on an FPGA. However, future technologies might make more favorable implementations possible (e.g. one could think of an analog architecture, which could realize the maximum more cheaply).

7.1.5. Conflict Detection and Commit

Conflict detection is performed as described in Section 6.3.2. As all metadata is stored in one cache line, the detection of read-after-write and write-after-write conflicts can be performed in the same cycle as the access. However, for the detection of write-after-read conflicts the read-set needs to be iterated. As it is possible that a read cache line was already evicted, this procedure can cause cache misses.

If no conflicts are detected, the write-set is confirmed. This is realized by iterating the write-set and setting the clean bit and timestamp of the affected cache lines. Again, it is possible for cache misses to occur. Additionally, all affected cache lines are also added to the cleanup queue.

If the core does not issue a request in a cycle, the cleanup queue is inspected. The oldest version is always in the first position, as the queue is FIFO. This version is compared to the versions in use by the trailing cores. If no trailing core is still busy confirming this or any older versions, it can be cleaned up. First, the data is merged to the safe version. This step has to be skipped if a newer version was already merged by another core. Then, the metadata and possible fallback cache lines are freed. Simultaneously, the cleanup queue is advanced.

The cleanup queue is not allowed to overflow. Additionally, a long cleanup queue increases cache misses, as these old cache lines might have been evicted. Therefore, the cache prioritizes cleanup once the length of the queue exceeds a configurable limit.

7.2. MicroBlaze Interaction

It is necessary for the MicroBlaze cores to interact with the HTM. Transactions have to be controlled. If the automatic mode is enabled, the cache takes care of this. Registers must be saved and restored, when a transaction aborts or for trailing execution. A customizable configuration makes it possible to optimize the system for the needs of a specific application. Performance counters give insights, which aspects influence the runtime.

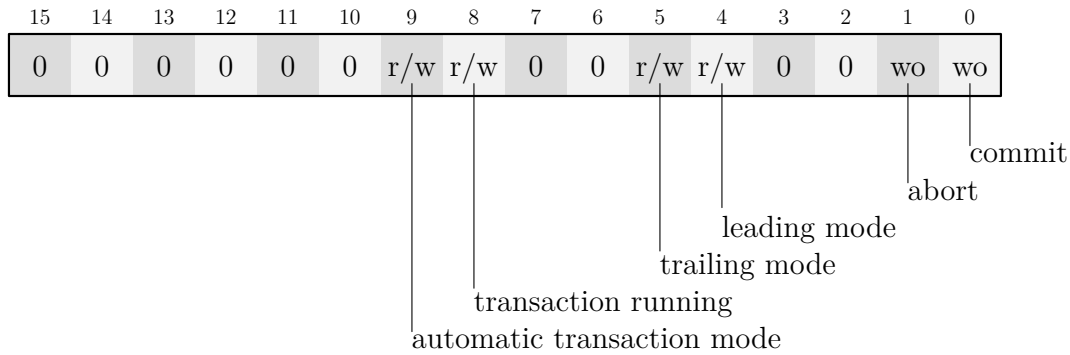


Figure 7.3.: The **Transaction Control Register** consists of multiple bits with separate functionality. The bits for commit and abort are write-only, as they are only used to invoke momentary actions. The other used bits can be both read and written. The unused bits return zero on read and should be set to zero on write.

7.2.1. Transaction Control

The MicroBlaze processor can control the transactions with a memory-mapped interface. Figure 7.3 shows the assignment of the **Transaction Control Register**, which is located at 0xF0000000. This register can be read to determine the current state of the transaction system. It can also be written to invoke actions or change the state. Every core can only access its own **Transaction Control Register**.

Common values, which are written to the **Transaction Control Register**, follow below:

- 0x101:** By setting the **transaction running** bit and invoking a commit type action a new transaction is started.
- 0x001:** Invoking a commit type action without the **transaction running** set causes the current transaction to commit.
- 0x102:** Invoking an abort type action causes an explicit abort. If there is no special code to handle the abort at the beginning of the transaction, the **transaction running** bit should be set, to ensure that the retry is executed in a transaction.
- 0x311:** Writing this value enables fault tolerance with the current core as a leading core. Both the **automatic transaction mode** and **transaction running** bit are set to ensure that all following code is executed in a transaction. This action is a commit type, as execution should continue with the next instruction after the write and a possible previous transaction should be committed before switching the mode.
- 0x322:** Writing this value enables fault tolerance with the current core as a trailing core. Both the **automatic transaction mode** and **transaction running** bit are set to ensure that all following code is executed in a transaction. This action

is an abort type, as execution continues at another location. The programmer has to ensure that the core has no pending transaction when this value is written.

7.2.2. Transaction Bounds

The transaction bounds can be determined explicitly or automatically. If transactions bounds are determined automatically, multiple limits can be applied. The limit, which is reached first, causes the transaction to commit and the next one to start. The metrics, which can be used as limits, are described below.

Runtime

The primary limit for the size of automatic transactions is their runtime. A uniform transaction runtime is desirable, as it results in good performance in redundant mode: Whenever a leading transaction finishes, the previous trailing transaction finishes at the roughly same time, as it targeted the same length. Therefore, validation of this transaction can start with no or small wait time. Additionally, the leading core can immediately free the speculative resources, which ensures that it never runs out of them.

This limit is realized by storing the timestamp, when the transaction is started. Every cycle the difference is compared to a configurable limit. When this limit is exceeded, the transaction is committed at the next possible opportunity. As our transaction system can only commit transactions at memory operations, the transaction can become longer than configured. This limitation could be resolved by changing the cores themselves. However, the used MicroBlaze cores are closed-source.

Executed Memory Operations

Another limit is the number of executed memory operations. For this limit, every memory operation (read and write) is counted. Once a configurable value is reached, the transaction is committed. This is a more consistent way to determine transaction bounds than runtime: Transaction runtime can fluctuate depending on the instructions, which are executed on other cores, as they can influence memory access latency. Additionally, memory operations limits are not negatively affected by the constraint that transactions can only commit at memory operations. Therefore, this limit is well suited to ensure that trailing transactions have exactly the same length as the corresponding leading transaction.

Read- and Write-Set

Transaction size can also be limited based on the size of the read- and write-set. The transaction commits once the size of one of them reaches a configurable value. This is useful, as it limits the hardware resources, which need to be available to store the read- and write-set. Additionally, this limit helps to constraint the number of cache misses per transaction. For example, a function that resets a memory area to zero will commit earlier than the runtime limit, as it quickly fills its write-set. A high number of cache misses in a single transaction is undesirable, as it can result in high fluctuations in trailing transaction runtime because cache miss latencies are variable. If the trailing transaction takes much longer than the leading transaction, the leading core might have to wait for it to finish, when the leading core runs out of speculative resources.

7.2.3. Configuration

The multiversioning system offers several configuration options, which can be set per core. However, it is recommended to use the same values on each core to ensure that trailing execution always behaves identically.

0xF0000004 Read-Set Limit: This setting limits the maximum size of the read-set.

It can be used to influence the size of automatic transactions. A high value results in slow commits, as every read-set entry needs to be validated.

0xF0000008 Write-Set Limit: This setting limits the maximum size of the write-set. It can be used to influence the size of automatic transactions. A high value results in slow commits, as every write-set entry needs to be marked as committed.

0xF000000C Optimizations: This value consists of multiple bits: Sticky Trailing Transactions (bit 5), Validate Automatic Read-Sets (bit 4), Use Fresh Fetch (bit 3), Cleanup During Miss (bit 2), Use Bloom Filter (bit 1), Use Compression (bit 0)

Each of these bits can be used to enable an optimization individually (see Section 7.4).

0xF0000010 Cleanup Low Limit: If this limit is exceeded, the cache will try to clean versions more aggressively even if it means slowing down execution. This can be advantageous, as a high number of versions can increase the cache miss rate.

0xF0000014 Cleanup High Limit: If this limit is exceeded, the cache will try to clean versions even more aggressively even if it means slowing down execution significantly. This limit should be used to ensure that the core never runs out of reserved fallback space.

0xF0000018 Runtime Limit: This setting limits the maximum execution time of an automatic transaction. It can be used to influence the size of automatic transactions. Trailing cores do not measure execution time themselves. Instead, they commit the transaction at the same instruction as the leading core.

0xF000001C Waiting Trailing Thread Queue Size Limit: This setting limits the maximum number of transactions waiting for validation on a trailing core. It prevents the leading cores from running to far ahead, which could negatively affect performance due to cache effects.

0xF0000020 Additional Miss Delay: This setting artificially increases cache miss latency. For optimum performance, it should always be set to zero. However, it can be useful to simulate the effect of different memory speeds on application runtime.

7.2.4. Register Backup

The trace port is used to back up the register set without impacting performance. It is also used for checksum calculation and to determine the instruction, at which to commit the current transaction. The signals, which are used for register set duplication and checksum calculation, are described below.

trace_valid_instr indicates whether a valid instruction is present at the trace port. Every instruction is signaled for one cycle. The MicroBlaze core can only commit a single instruction per cycle.

trace_msr_reg matches the current value of the MSR. In particular, this register contains the carry, which is needed if an automatic transaction commits between two instructions using the carry.

trace_pc contains the PC of the committed instruction

trace_instruction outputs the committed instruction. It is mainly used to detect the special **imm** instruction, which is merged with the next instruction to support longer immediates.

trace_reg_write indicates whether a register was written.

trace_reg_addr contains the index of the written register.

trace_new_reg_value contains the new value written to the register. This is the primary signal used to calculate the checksum and also important for register set backup.

trace_jump_taken indicates whether the current instruction is a jump or taken branch.

trace_delay_slot indicates whether the current instruction was executed in the delay slot [73, p. 58] of a previous branch.

trace_data_access signals whether the current instruction accesses the memory. It is required to find transaction start instructions on the trace port to snapshot the register set at the correct time.

trace_exception_taken signals if an exception or interrupt is taken.

trace_exception_kind contains the kind of exception issued. It can be used to rollback immediately if a bit-flip causes the core to access an invalid memory location.

trace_mb_halted indicates whether the core is halted by a debugger. The debugger can halt the core spontaneously (e.g. to collect performance data) even in the middle of a redundant transaction. Therefore, all checksum calculations, watchdog timers and transaction limits have to be suspended while the debugger is active.

There are other signals, mainly concerning speculative structures like the branch target buffer or the disabled internal caches. However, these were not used in this implementation.

In addition to the register set on the core itself, there are two storage areas for the registers in the cache. The first one replicates the current register values. Whenever **trace_valid_instr** and **trace_reg_write** are active simultaneously, the value of the register **trace_reg_addr** is updated to **trace_new_reg_value**. Some register updates (e.g. when an interrupt changes two registers at once), cannot be recorded, correctly. However, none of these cases should occur in a correctly functioning user space application. The second storage area serves as register backup. It is updated in a single cycle by copying from the first area, when a new transaction is started. If a transaction is aborted, these values are restored by the interrupt handler (see Section 7.2.5).

It is important to keep exact count of the memory operations in both cache and trace interpretation. This is necessary to ensure that the register snapshot is taken at the correct memory operation, which intended to begin the transaction. There is no reliable way to correlate memory operations if synchronization is lost, as there is no instruction counter. Simple ways like looking at the PC fail, as an automatic transaction can also commit in a small loop. For example, imagine a loop spinning on a lock. All memory accesses within this loop except for the last one will look exactly the same. However, the memory access to commit at has to be identified exactly to ensure matching checksums. This is necessary, even though the multiversioning system ensures identical reads, as adding the same value a different number of times to a checksum still changes it.

There are some constraints when a transaction can commit:

- A transaction can only commit at a memory operation, as it is necessary to stall the core, while the read-set is validated and the write-set marked as clean.
- *Imm* is a special instruction, which extends the immediate of the next instruction [73, p. 251]. A transaction cannot commit right after an *imm* instruction, as its effect cannot be restored in an interrupt handler. Therefore, the state before the *imm* instruction needs to be saved if a commit is attempted at that point.
- Transactions cannot be committed in a branch delay slot, as the target of the branch is not known. If this case occurs, the state before the branch has to be stored, so that the branch is also repeated if the transaction is aborted.

The limitation that transactions cannot commit in a branch delay slot causes complications, as some branches also modify registers. If the branch also depends on this changed register, an identical repetition cannot be guaranteed. There are three branch instructions that can cause this: *brld*, *brald*, *breald*. These branches all store the current PC in a register and all use a branch delay slot. Additionally, they branch to an address, which is determined by a register. They only differ by the calculation of the address (relative/absolute) and width of the registers. It seems as if compilers do not generate problematic branches, where the source and destination registers are the same, as regular interrupts would suffer from similar issues.

7.2.5. Transaction Abort

To abort a transaction, it is necessary to restore both memory and processor state. The memory rollback is performed by the data cache. It iterates the write-set and deletes all new versions. The cleanup queue is not necessary for this step, as metadata and fallback cache lines are freed immediately. The read-set is cleared without iterating it, as reading a cache line does not change any multiversioning metadata or versions.

To perform the rollback on the cores, an interrupt is issued. The interrupt might not occur immediately. Therefore, the core can still issue some memory accesses. To avoid exceptions like jumps to non-executable sections, the cache returns the actual memory values for reads. Writes are dropped. Usually at most two memory operations are performed: The one, which causes the abort, and another one, which is already in the pipeline.

The interrupt handler then restores the processor state. First, the interrupt handler restores the MSR. This is performed first, as an unexpected path, which was taken because of a race condition, could have toggled problematic bits like the *MMU enable*. Then, most registers are restored. The first read of the saved registers also

notifies the cache that it should no longer consider memory writes as part of the aborted transaction. The saved PC is stored in R14, which is the interrupt return address. A simple backoff mechanism is performed next. The core reads the current cycle count and takes the last 10 bits. It then spins for that many iterations. A better implementation in hardware like we presented in [46] would be favorable, but is not used in the current version. Finally, the interrupt returns. The delay slot of the jump is used to restore one more register. This leaves R14, which cannot be restored, as the memory-mapped region cannot be addressed with an immediate and there is only one delay slot available. However, this register should not be used by a user mode application, as it is reserved for the interrupt return address.

The information that is required for the abort itself and its handling is provided by a memory-mapped interface.

0xF0001000 R0 - 0xF000107C R31: This region contains the values of all registers at the last transaction start. Note that R0 and R14 are not read by the interrupt handler, as R0 is always zero and R14 is the interrupt return address.

0xF0002000 PC: This memory address contains the PC at the last transaction start.

0xF0002004 MSR This memory address contains the MSR at the last transaction start.

0xF0002008 Last Abort Cause The value at this address specifies the cause of the last abort. The following values are possible:

- 0: No abort has occurred.
- 1: This is a trailing core starting its next trailing transaction.
- 2: An explicit abort was issued by the application.
- 3: A read-after-write conflict has occurred. This core is the reader.
- 4: A write-after-write conflict has occurred. This core is the second writer.
- 5: A write-after-read conflict has occurred. This core is the reader.

This value is primarily used for debugging.

0xF000200C Last Abort Address The value at this address specifies which memory address was accessed to cause the abort. As the conflict detection works at cache line granularity, this value is also accurate at cache line granularity. This value is also correct if another core causes the conflict. This value is primarily used for debugging.

0xF0002010 Consecutive Abort Count This value specifies how many aborts have occurred since the last successful commit. It is used to switch to another thread if too many aborts occur consecutively.

0xF0002014 Virtual Core Id This value specifies the id of the current core for leading cores and the id of the corresponding leading core for trailing cores. It is used by some synchronization constructs.

7.2.6. Trailing Execution

On the MicroBlaze side trailing mode is entered by writing 0x322 to the control register (see Section 7.2.1). This aborts the current transaction (if available) and enters the abort handler. The core then waits until an actual transaction becomes available.

The management of trailing execution is handled by the caches. There is a FIFO queue, which contains all transactions, which are currently waiting for execution. Each queue entry contains all data that is necessary to start the trailing transaction. Primarily, the values of all registers including PC and MSR are available. These are then used by the abort handler of the trailing core. Additional information, which is needed to ensure an identical execution, like the core id or the number of previous aborts is also applied.

The queue is used to transfer the number of executed memory operations in the leading transaction to the trailing cache. The trailing cache then only uses this information to determine the transaction length. Transaction management operations are discarded or considered a mismatch depending on the specific operation. Once the correct transaction length is reached the checksum calculated in the trailing core is compared to the one included in the entry. If they differ, rollback is initiated. In the error-free case, the trailing core can continue to confirm the next transaction. This is realized by aborting the current transaction, which invokes an interrupt that starts the process of fetching the next register set again. Additionally, all versions created by the trailing core are deleted. The versions created by the leading core will later be merged to the safe version by the leading core itself.

7.2.7. Performance Counters

The system provides various performance counters, which are memory-mapped in the region 0xF0003000 to 0xF000323B. Performance counters are reset, when the core is reset. A reset by software is not possible. Therefore, an application needs to store the values of the performance counters at the start of the region of interest. At the end of the region of interest, these stored values can be subtracted from the current values to get the measurements for the region of interest only. However, this is only possible for incrementing performance counters.

Our implementation uses three kinds of performance counters:

Constant: Constant performance counters return static values determined at compile time. They are mostly used to correctly correlate other performance counters under the presence of context switches and to convert their unit. The cost of a constant performance counter is very low in regards to area and frequency overhead.

Dedicated: Dedicated performance counters consist of an up to 64 bit register, which can be read by the application. Arbitrary operations (e.g. increment, max) can be performed on this register. The performance counter can either be associated to a single core or shared between all cores. The cost of a dedicated performance counter is high in regards to area, especially if it is associated to a single core. It can also limit the frequency of the design if the performed operation is expensive.

Optimized: All optimized performance counters of one core share a single structure called `SPIN_STATS`. This structure cycles through all performance counters periodically. Therefore, read operations take multiple cycles. Only non-negative increments can be performed and their magnitude is limited to the number of memory controllers each cycle. These increments are stored in a narrow register until the counter becomes active. This makes it possible to use a single shared wide adder for all optimized counters of one core. Optimized performance counters cannot be shared between cores. The cost of an optimized performance counter is relatively low in regards to area and frequency overhead.

Table 7.1 shows how the performance counters are mapped into memory.

Various counters are available to assess the performance of the caches themselves. There are counters, which measure the number of reads or writes. Counters for the number of cycles taken by these operations are also available. In combination, these counters can be used to determine the average access latency. The number of cache misses and writebacks are also available.

Counters for the transaction operations fence (begin and commit atomically), abort, begin and commit are available. Additionally, the time spent to commit transactions is also measured. The counters for useful cycles and wasted cycles can be used to determine whether there is an issue with conflicts. All cycles inside a transaction, which ultimately commits, are considered useful. Contrary, all cycles inside a transaction, which ultimately aborts, are considered wasted. The number of allocated fallbacks and time spent freeing them (called *Pause Cycles*) is also available. For the fallback counter, both a variant that counts every version using a fallback (called *Fallbacks*, usually large) and the number of allocated cache lines in the fallback region (called *Allocated Fallbacks*, usually small) exist. These counters can be used to fine-tune the relevant configuration values.

There are also counters in regards to fault tolerance. The time that was spent waiting for a free version is measured. A large value is an indicator that the trailing cores take

Address	Type	Size	Description
0x3000	Optimized	53 bit	D-Cache Reads
0x3008	Optimized	53 bit	D-Cache Misses
0x3010	Optimized	53 bit	D-Cache Writes
0x3018	Optimized	53 bit	Fallbacks
0x3020	Optimized	53 bit	D-Cache Writebacks
0x3028	Optimized	53 bit	I-Cache Reads
0x3030	Optimized	53 bit	I-Cache Misses
0x3038	Optimized	53 bit	Transaction Fences
0x3040	Optimized	53 bit	Transaction Aborts
0x3048	Optimized	53 bit	Transaction Begins
0x3050	Optimized	53 bit	Transaction Commits
0x3058	Optimized	53 bit	D-Cache Read Cycles
0x3060	Optimized	53 bit	D-Cache Write Cycles
0x3068	Optimized	53 bit	I-Cache Read Cycles
0x3070	Optimized	53 bit	Instructions
0x3078	Optimized	53 bit	Pause Cycles
0x3080	Optimized	53 bit	Transaction Commit Cycles
0x3088	Optimized	53 bit	Waiting For Version Cycles
0x3100	Constant	32 bit	D-Cache Size
0x3104	Constant	32 bit	D-Cache Associativity
0x3108	Constant	32 bit	Cache Line Size
0x310C	Constant	32 bit	Physical Core Id
0x3110	Dedicated (shared)	48 bit	Cycle Count
0x3118	Constant	32 bit	I-Cache Size
0x311C	Constant	32 bit	I-Cache Associativity
0x3120	Dedicated (shared)	32 bit	Global Rollback Count
0x3200	Dedicated (private)	48 bit	Total Useful Cycles
0x3208	Dedicated (private)	48 bit	Total Wasted Cycles
0x3210	Dedicated (private)	32 bit	Allocated Fallbacks
0x3218	Dedicated (shared)	48 bit	Max Fault Detection Latency
0x3228	Dedicated (shared)	64 bit	Sum Fault Detection Latency
0x3230	Dedicated (shared)	48 bit	Sum All Checksum Modification Count
0x3238	Dedicated (shared)	32 bit	Current All Checksum Modification Count

Table 7.1.: The mapping of performance counters into memory is shown above. All addresses are relative to the base address (0xF0000000). Depending on their size, either 4 or 8 bytes are dedicated for one counter, even if this results in some unused bits. This enables easy access with regular integer operations.

too long to confirm transactions. The number of global rollbacks due to checksum mismatches is also counted. Various counters measuring fault detection latency are available. They are used for the evaluation in Section 9.3.

General counters like the number of executed instructions and elapsed cycles are present. Additionally, general system information like the size and associativity of the caches can be retrieved. Note that the *Physical Core Id* is constant and not affected by trailing execution. Therefore, it should not be used for synchronization, but rather only for debugging.

7.3. Devices

In addition to the cores and their caches, there are also other devices available: Two memory controllers provide adequate bandwidth to run benchmarks on 12 cores. A UART module provides input and output capability. Several hardware debuggers simplify the development of multiversioning and the porting of software.

7.3.1. Memory Controllers

Two DDR4 memory controllers provide an AXI interface to one 4 GB DDR4 component memory each. Only a total of 2 GB is accessible by the application due to the limited address space. The memories run at 1200 MHz (DDR4-2400) with a CAS (Column Address Strobe) latency of 15 ns. Xilinx Memory IP [80] is used for communication with the DDR4 modules. The controller runs at 300 MHz with a bus width of 512 bit. However, this cannot be fully utilized, as `AXI_MEM0` and `AXI_MEM1` run at the core frequency of 50 MHz and have a reduced width of 128 bit. We added an additional component to reset all memory to zero, when the cores are reset, as leftover version information from the last run could confuse the multiversioning logic.

7.3.2. UART

All cores can access a UART module, which is connected to the `AXI_DEV` port. The MicroBlaze Debug Module [72] provides this UART functionality. Data that is output to the UART module is sent to the host machine over the JTAG connection.

7.3.3. Debuggers

The following hardware debuggers are available:

Integrated Logic Analyzer: The Integrated Logic Analyzer [70] is an IP provided by Xilinx and is used to inspect some signals. It can record a trace of 1024 cycles, when a configurable condition is met. It is connected to the memory interfaces and trace port of the third core. `AXI_MEM0` can also be inspected. Additionally, some dedicated debug signals are available.

Custom Assertions: As Vivado does not synthesize SystemVerilog assertions, we have implemented our own assertion system. The interface is provided by a macro, which accepts the condition and optionally the core id as arguments. When an assertion is triggered, the corresponding line number is output to the ILA. If multiple assertions trigger at the same time, only the smallest line number is output. The output is delayed by one cycle to reduce timing impact. Execution resumes even after an assertion is triggered.

MicroBlaze Debug Module: A MicroBlaze Debug Module [72] is connected to all cores. This enables debugging of the application running on the MicroBlaze cores in Eclipse. Additionally, it is also used to upload the application binary to the FPGA and to download the output files. The UART interface is also provided by this module.

DDR4 Memory Controller: Vivado also provides debugging support for the DDR4 memory controllers [80]. However, this does not include the accessed addresses or memory values. As the board comes already populated with the correct memory, this debugger is largely useless for us.

7.4. Optimizations

Various optimizations can be implemented to improve the performance of the multiversioning system.

7.4.1. Validation of Automatic Transactions

By default, every transaction performs full conflict detection. If a conflict is detected, all but one of the conflicting transactions are restarted. For automatic transactions, this is not necessary, as the same code works without conflict detection in the non-transactional implementation of the PARSEC benchmarks. It is not possible to disable write-after-write conflict detection, as this would make complicated merges of cache lines that were changed by two or more cores simultaneously necessary. Read-after-write conflicts already offer good performance due to the optimizations

described in Section 6.3.3. Therefore, only write-after-read conflicts can profit from additional optimizations.

For most benchmarks, write-after-read conflicts can simply be ignored. This does not only reduce overhead from the repeated execution of aborted transactions, but also from conflict detection itself. However, if a benchmark contains race conditions, these can cause trouble with redundant execution. If the order of the trailing execution is swapped, the trailing cores might now return a different result. This then causes a rollback of the whole system. Contrary, if write-after-read conflicts are handled on the leading cores, only a single core has to roll back for each conflict. Note that a write to a single cache line can still cause conflicts with multiple reading cores, which can result in additional rollback even if the conflicts are handled on the leading cores.

7.4.2. Bloom Filter

Another way to reduce conflict detection overhead is the use of a Bloom filter [17]. A Bloom filter is a constant size, probabilistic data structure implementing a set. It is possible to add memory addresses to the set and query whether they are contained in the set. However, the more addresses are added, the higher the probability is that a query falsely returns that the address is contained in the set (false positive). It is not possible for a query to falsely return that an address is not contained (false negative). Therefore, this data structure is well suited for conflict detection, as a false conflict only costs performance, while a missed conflict could cause wrong outputs.

In our implementation, the Bloom filter is used to optimize write-after-read conflict detection. Every core has its own Bloom filter. When a transaction reads a cache line, its address is added to that core's Bloom filter. When a transaction writes a cache line, every other Bloom filter is queried for its address. If a Bloom filter returns that it contains the address, regular conflict detection and resolution is performed for the corresponding core. This optimization reduces the number of cache misses and the overhead for commits, as it is not necessary to iterate the read-set if the Bloom filter did not indicate any hits. The Bloom filter is reset on commit or abort of the transaction on the corresponding core.

The implementation used in the FPGA is kept simple, as the number of accessed cache lines in each transaction is rather small, and fast lookups are required. Therefore, only a single hash function is calculated for each entry. The corresponding bit in the constant size memory is set to one, when the entry is inserted. To query whether the entry is contained, it is sufficient to return the corresponding bit. If the allowed transaction size is increased, a more complex Bloom filter, which uses multiple hash functions, should be used. In such a Bloom filter, an entry is only considered contained if the corresponding bits of all hash functions are one. Using the optimum number of hash functions results in a minimal false positive probability.

7.4.3. Cache Line Compression

In the basic variant, an additional cache line is allocated for each new version of a cache line. This can be optimized by directly storing the changed data in the field for the address of the version. This optimization is possible if at most eight bytes have changed. The change has to consist of two aligned four byte words, one at an even index (3rd address bit equals 0) and one at an odd index (3rd address bit equals 1) in the cache line. Permitting this split increases the likelihood that a stack cache line can be compressed. Because of the index requirement, only one comparison is required per memory access, as all MicroBlaze accesses are four byte aligned and the bus is four bytes wide. If more data is written, the additional cache line is allocated and the existing changes are transferred to free the address field.

The mode & index field is used to store the necessary metadata. Mode is a field consisting of 2 bits per version, which indicate whether the high and low words use compression, respectively. If both bits are zero, the address points to a regular fallback cache line instead. Index consists of two 3 bit integers, which indicate the indexes of the overwritten words. To simplify the implementation, it is shared between all versions. Additionally, a version is not allowed to use compression if a previous version uses a fallback cache line. This avoids complicated partial writes, which would need to fetch data from another cache line to write it back to the initial one.

This optimization reduces the number of cache misses, as the cache miss for version allocation is avoided. Additionally, fewer cache lines are evicted prematurely. The optimization also reduces the overhead of reading or writing a version that is not the safe version. If the required version is stored in another cache line, at least one additional cycle is required to access it. This overhead is avoided, as the address field can be accessed in the first cycle.

7.4.4. Fresh Fetch

When a new version is allocated, the data of the cache line is copied to a fallback cache line. This means that a cache line, which has not been accessed for a long time, is written. In most cases, this causes a cache miss. This optimization avoids the overhead of the actual fetch by clearing cache lines to zero instead of fetching them if they will be completely overwritten. This is always the case for version allocations. Note that the overhead for the writeback of dirty cache lines can still occur.

7.4.5. Sticky Trailing Threads

When a transaction commits on the leading core and multiple trailing cores are ready, the trailing core that confirms the transaction is selected arbitrarily. If sticky trailing threads are enabled, trailing cores, whose previous transaction was from the same leading core, will be preferred. Most of the time, one trailing core will stick to validating exactly one leading core. This reduces cache misses, as the trailing core's cache already contains some reused cache lines from the last transaction, e.g. the stack.

7.4.6. Important Writes

The data written by trailing cores does not need to be preserved for further execution, as all later transactions use the leading core's version. For fault detection, it is included in the checksum, but the actual changed value is not stored. We assume the cache and main memory are protected by other means like ECC. Therefore, the data written by the trailing core is not needed to ensure their correctness.

However, sometimes the data is needed to correctly execute the trailing transaction. Assume a case where a transaction first writes a location, then reads it and finally overwrites it. The read value is stored in neither the safe version nor the leading version. Therefore, the trailing core has to actually perform these stores.

We use a 128 bit Bloom filter with a single hash function to detect these stores. When the leading core reads a cache line, which is already contained in the write-set, it adds its address to the Bloom filter. The Bloom filter is transferred to the trailing core together with the registers. The trailing core then checks this Bloom filter on every write and only performs it if the address is contained. This ensures that every write, which can influence the rest of the transaction, is actually performed. The probabilistic nature of the Bloom filter can only result in unnecessary writes, but never missed writes.

This optimization reduces cache misses in the trailing core, as cache lines, which are written but never read, are not fetched. Additionally, less cleanup is required at the end of the trailing transaction, as less versions were created. Therefore, the trailing core is ready to execute the next transaction more quickly, which results in less waiting time on the leading cores.

Bit-flips can still be detected reliably with this optimization enabled. Faults in the written data and address are detected, as they are part of the checksum. If the Bloom filter is corrupted and an important write is omitted, the later read will return a different value on the trailing core than on the leading core and the checksums will mismatch. If the written data is identical to the data already in memory, a corrupted Bloom filter might not be detected. However, the resulting state is correct.

An unnecessary write caused by a corrupted Bloom filter only costs performance, but does not corrupt memory.

7.5. Summary

We implemented our fault tolerance with multiversioning approach on an FPGA. The fact that this implementation was possible and works well shows that there are no major oversights in the approach. We use the MicroBlaze soft cores by Xilinx as processors. Changes to the cores were not necessary. Instead, a custom cache implementation provides multiversioning functionality. The necessary interaction with the cores is handled by a memory-mapped interface, the trace port and interrupts. Several hardware structures in the cache have to be extended. In the FPGA implementation, this is rather expensive: Double the space and more than double the comparators are required. However, there is still potential for optimization by making better use of hardware characteristics.

The memory-mapped interface serves several purposes: A register is available to control transactions and redundancy. Additional settings are also available. Rollback is invoked by an interrupt and the handler then restores the processor state by reading register values from cache. The invocation of the interrupt and backup of the register set is handled by the cache. The cache is also responsible for managing the trailing execution. Lastly, performance counters are available in a memory-mapped interface.

In addition to the caches, other devices are also available: Two interleaved memory controllers provide sufficient memory bandwidth. A UART module handles input and output. Hardware debuggers simplify development.

Several optimizations can be switched on to improve performance: The validation of automatic transactions is often unnecessary and can therefore be disabled. A Bloom filter reduces the impact of read-set validation. The overhead of allocating new versions can be reduced by omitting the fetch if the cache line is overwritten entirely. Validating transactions from one leading core on the same trailing core can improve cache locality. Omitting unnecessary writes on the trailing core reduces the number of versions that need to be allocated.

Altogether, this FPGA model of multiversioning is well suited to evaluate the approach. Thanks to the compiler support and sufficient performance, real workloads can be run to verify correct functionality. The availability of 12 cores enables scaling analysis. The trailing execution makes it possible to evaluate the impact of fault tolerance on the runtime. The performance counters can be used to measure error detection latency.

8

Porting the PARSEC Benchmarks

We ported the PARSEC benchmark suite [15] version 3.0, since we use it for our evaluation. It is executed without operating system on our FPGA prototype. Several APIs had to be implemented, as they are required by the PARSEC benchmarks. We use a specially developed launcher to execute the benchmarks in batch. Furthermore, we ported two variants of the pthreads library using atomic operations and transactions respectively. Handling for atomic operations in multiversioning mode is also available. Some changes to the benchmarks were necessary to compile them for the MicroBlaze architecture and to ensure that they run reliably.

Table of Contents

8.1 Bare Metal Execution	94
8.2 Launcher Tool	96
8.3 Pthreads Implementation	97
8.3.1 Thread	97
8.3.2 Mutex	98
8.3.3 Barrier	100
8.3.4 Once	101
8.3.5 Scheduling	102
8.3.6 Condition Variable	102
8.3.7 Thread Local Storage	102
8.3.8 Other	103
8.4 Atomic Operations	103
8.4.1 Atomic Operations in Transactions	104
8.4.2 Conversion of Atomic Operations	104
8.4.3 Fallback for Uncaught Load Linked/Store Conditional	106
8.5 Benchmark Details	107

8.6 Summary	107
-----------------------	-----

8.1. Bare Metal Execution

We run the PARSEC Benchmarks bare metal without operating system. As the benchmarks were not written for this mode of operation, some interfaces have to be provided:

Memory Management: The benchmarks make use of dynamic memory allocation. The compiler already provides this functionality through a heap in a static memory region. This is sufficient for the execution of single benchmarks. To run multiple benchmarks at the same time, it is necessary to provide multiple binaries with different memory regions, as there is no MMU or relocation. Note that this is required anyways, otherwise the code of the first benchmark would be overwritten by the second benchmark. We modified the memory management functions to be thread safe by encapsulating them in a critical region.

Input/Output: Output to a UART port is already provided by the included Xilinx libraries. We modified the functions to be thread safe by encapsulating them in a critical region. The benchmarks do not require interactive input. However, process arguments are used to select work packages and thread counts. We define the arguments in source code, which causes them to be compiled into the binary. This avoids the need for a launcher process. Constants can be used to quickly change parameters of all benchmarks at the same time by modifying a single file. However, recompilation of all benchmarks is required.

File System: Some benchmarks read their input data from files and write their results to files. We provide this functionality with a memory file system. The input files are compiled into the binary by an assembler file. Changed files are stored in memory. The address and length are output to the UART port and the file can then be downloaded to the host system by the debugger.

Some of the required functions are undefined in the standard library included with the MicroBlaze compiler, although the C++ specification requires them. We simply provided an implementation for them. Other functions are already defined in the included standard library, but do not fulfill the requirements. In these cases, we make use of the `-wrap` linker flag [40] to replace them. Note that in most cases our implementation does not completely fulfill the requirements of the specification, either. However, they are sufficient for the execution of the benchmarks. Table 8.1 shows which benchmarks require certain functions. Appendix A describes the functions in detail.

Function	Implementation	Benchmarks
File System		
open	full	all except swaptions
close	full	all except swaptions
read	full	all except swaptions
write	full	all except canneal & swaptions
stat	limited	dedup, facesim, ferret, vips
fstat	limited	vips
lseek	full	vips
access	stub	raytrace, vips
opendir	full	ferret
closedir	full	ferret
readdir	full	ferret
mmap	limited	vips
getcwd	stub	vips
ftruncate	full	vips
link	crash	x264
Memory Allocation		
malloc	wrapper	all
calloc	wrapper	all
realloc	wrapper	all
free	wrapper	all
Compiler Helper		
--cxa_guard_acquire	full	bodytrack, canneal, facesim, fluidanimate, raytrace, streamcluster, swaptions, vips
--cxa_guard_abort	full	bodytrack, canneal, facesim, fluidanimate, raytrace, streamcluster, swaptions, vips
--cxa_guard_release	full	bodytrack, canneal, facesim, fluidanimate, raytrace, streamcluster, swaptions, vips
Various		
printf	calls vprintf	all
vprintf	wrapper	facesim, vips
gettimeofday	limited	facesim, ferret, raytrace, vips, x264

Table 8.1.: Some functions had to be provided or modified for the execution of certain benchmarks. While some functions were implemented fully besides error conditions, other provide only the subset of the functionality, which is required by the benchmarks. Two functions are only stubs, returning zero or empty, and one function would crash with an error message if it was called. For functions, which only required additional synchronization, wrappers are used.

8.2. Launcher Tool

We have developed a launcher running on the host machine to simplify the evaluation. The launcher can start all benchmarks in batch mode with multiple iterations per benchmark possible. If desired, faults are injected. Their output is validated and performance data like runtimes is collected.

Before starting the launcher, the FPGA has to be programmed with the bitstream containing the MicroBlaze cores and the multiversioning enabled caches. Therefore, the MicroBlaze Debug Module is already available. Our launcher communicates with it by using the TCP/IP (Transmission Control Protocol/Internet Protocol) interface of *XSDB* [82], which in turn uses a virtual JTAG port over USB. The FPGA is not reprogrammed to start a new benchmark. Instead, a reset signal is asserted to all components. Then the `down` command of *XSDB* is used to transfer the application binaries. As the startup logic is provided in separate binaries for each core, it has to be invoked for every core. After successful transfer of the binaries, the `con` command is used to start execution.

During execution, a bit-flip can be injected (see Section 9.4). As the VIO is not accessible by the *XSDB*, a `.tcl`-script is used instead. Our launcher communicates with it by writing files at special locations.

UART output is collected by *XSDB*. We use the `jtagterminal` command to receive the output and write it to the host file system. This output also contains pointers to all files written by the benchmark. After the execution has finished, we copy these files to the host system using the `mrd` command.

Once all output data has been collected, the execution is validated. First, UART output is matched to a benchmark-specific regular expression. This regular expression contains patterns for changing parts of the output, like pointers or runtimes. The benchmark *canneal* requires special handling, as the relative difference of its floating point output needs to be calculated, which cannot be done in a regular expression. Next, the files are compared to reference outputs created by x86 execution. For the benchmark *x264*, the output file depends on the thread count. Therefore, we provided reference files for all used thread counts. The benchmark *vips* writes the current time to the output file. We ignore it when comparing them to the reference file. The output of the benchmark *ferret* is in arbitrary order. Therefore, we sort it before we compare it to the reference file.

After output validation, we use regular expression to extract performance counter values from the UART output. Additionally, the number of global rollbacks can be read from the VIO. This is useful if the benchmark froze as result of the injected bit-flip and performance counter values are not available. The performance counter values are aggregated and output to a `.csv`-file together with the validation result. Finally, the next iteration of the same benchmark or the first iteration of the next

benchmark is started. After all benchmarks have finished, TCP/IP connections and files are closed before the launcher exits.

The launcher contains various precautions to deal with misbehaving benchmarks. A watchdog timer terminates benchmarks, which have frozen. The timeout interval depends on the benchmark. A benchmark flooding the UART port is detected and terminated. The UART port is flushed during reset to avoid garbage output at the beginning of the next benchmark.

8.3. Pthreads Implementation

We have implemented a subset of the pthreads library. There is both an implementation using atomic operations and using transactional memory. As there is no kernel and no timer interrupts, cooperative multitasking is used instead of preemptive multitasking. All synchronization constructs switch threads, when they block for too long.

For this section, we assume that the reader is familiar with the pthreads library. The specification is available at [32]. The Linux man pages [49] also contain detailed descriptions of the pthreads library.

8.3.1. Thread

Two separate data types are used to manage threads. The first, called `thread_info`, contains the data needed for the pthreads functions themselves. This is the type `pthread_t` points to. It consists of the return value `retval`, the stack pointer `stack_pointer`, thread local storage `specific`. It has a `status`, which can be `UNINITIALIZED`, `WAITING_FOR_START`, `RUNNING`, `WAITING_FOR_JOIN` or `FINISHED`. All performance counters are also stored in this structure. They are decreased by the value of the hardware performance counter when the thread is switched in and increased when it is switched out. Therefore, in total each performance counter variable is incremented by the change of the hardware performance counter. This ensures that performance counters are measured separately for each thread and only while the thread is running.

The second data structure `suspended_thread` contains the state necessary to switch to a thread. For the MicroBlaze cores, this state consists of the registers, the PC and the MSR. The current thread is stored in a global variable at a core dependent address. Therefore, it is not necessary to include it in the processor state. A ring buffer is used to queue suspended threads. When a thread is created or suspended, it is added to this buffer. Idle cores poll this buffer to resume a thread.

The `pthread_create` function creates a new thread. It first allocates all necessary structures including `thread_info`, `suspended_thread` and the stack. The register

set is initialized as needed by a special starter function. This starter function changes the status to `RUNNING`. It also calls the intended function and the `pthread_exit` function if necessary. However, the current core does not immediately switch to this function. Instead, the thread's status is set to `WAITING_FOR_START` and the corresponding `suspended_thread` is enqueued in the ring buffer for suspended threads. The `pthread_create` function also writes a magic value to specific memory address. This memory address is validated by other cores before they start executing to ensure that initialization of the system has completed.

The `pthread_join` functions waits in a loop for completion of the target thread. Every 1000 iterations a context switch is attempted. When the target thread's status changes to `WAITING_FOR_JOIN`, it saves the return value and outputs the performance counters. Then the data structures and stack are freed. Finally, the thread's status is set to `FINISHED`.

The `pthread_yield` function attempts a context switch. If the ring buffer of suspended threads is empty, it returns immediately. The `pthread_yield` function is nonstandard, but is present on many systems [48]. For our implementation, it is required to prevent deadlocks, as no periodic preemption is available.

The function `pthread_self` returns the current thread by reading the global variable corresponding to this core. If this variable is `NULL`, the negated core id is returned instead. This value cannot be used with most functions, but is sufficient for early initialization like the allocation of static variables.

The function `pthread_equal` compares two threads by checking their pointers for equality.

The function `pthread_detach` is not implemented and not used by any PARSEC benchmark.

The `pthread_exit` function calls the destructors for all thread local variables (see Section 8.3.7). The return value `retval` is set to the parameter and the performance counters are updated one last time. The `status` is set to `WAITING_FOR_JOIN`. Finally, the active thread is cleared and the core enters its idle loop. Note that memory cleanup is the responsibility of the thread calling `pthread_join`.

The `pthread_attr_` group of functions is only implemented as stub, as the underlying functionality is not implemented and not required by any PARSEC benchmark. However, the benchmarks *fluidanimate*, *swaptions* and *vips* call these functions with their default values, which makes it necessary to define these functions.

8.3.2. Mutex

Our implementation provides recursive mutexes that can be locked multiple times by the same thread. Each mutex stores its current owner and a recursive lock count.

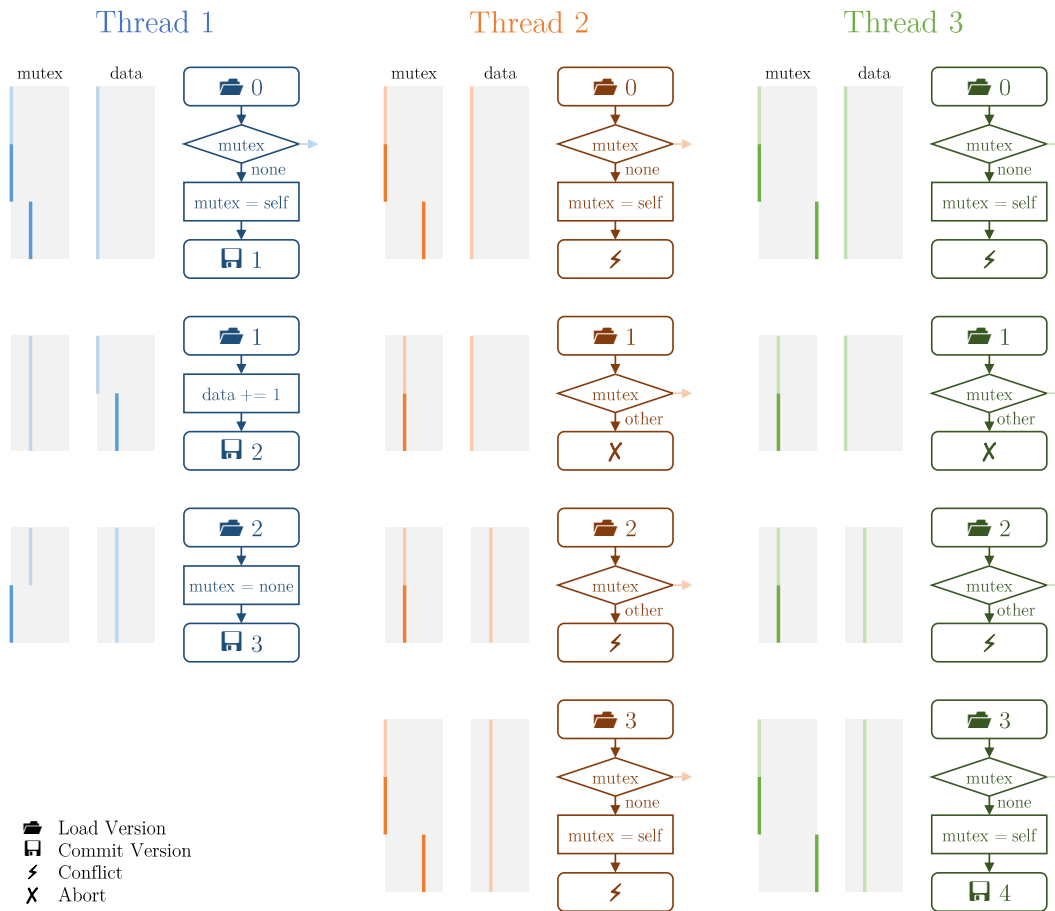


Figure 8.1.: The program shown here consists of three threads, which protect a critical section with a non-recursive mutex. The threads operate on the same memory, but their current view (gray boxes) can differ, since writes only propagate on commit. The lines of memory locations, which are added to the read- or write-set, are marked in a darker color. Every thread first locks the mutex, then increments data by 1 and finally unlocks the mutex. Initially, all threads appear to successfully lock the mutex, as the mutex is free in version 0. However, when attempting to commit, only one thread can succeed (here thread 1), as they have all read and written the same memory location. Therefore, the other threads roll their transactions back. Now thread 1 has locked the mutex and this state is also globally visible. Thus, it can execute its critical section. The other threads now read version 1, in which the mutex is locked and abort. After completing the critical section thread 1 unlocks the mutex again. The other threads only notice this after the transaction encapsulating the unlock operation is committed. Afterwards, they again compete for the mutex. This time thread 3 wins.

For non-recursive mutexes, the recursive lock count is not needed.

The `pthread_mutex_init` function initializes all attributes to zero, resulting in an unlocked mutex. The `pthread_mutexattr_` function group is implemented as stub only, as the PARSEC benchmarks only use the default values. We did not implement the `pthread_mutex_destroy` function, as it is not used by the PARSEC benchmarks.

The `pthread_mutex_lock` first checks whether the current thread, which is determined by calling the `pthread_self` function, is the owner of the mutex. If this is the case, the recursive count is incremented and the function returns immediately. Otherwise, the function waits in a loop until the owner is zero. The owner is then set to the current thread atomically or in a transaction respectively. The recursive lock count is also set to one before returning. A context switch is attempted every 10,000 iterations of the loop.

The `pthread_mutex_unlock` first decrements the recursive lock count by one. If the count is zero, the owner is also set to zero. The function returns without waiting.

Figure 8.1 shows the transactions that occur when executing a critical section with multiversioning. Note that the loop iterations do not generate unnecessary versions, as their transactions abort. In redundant mode, multiversioning ensures that the leading and trailing threads read the same values and the mutex logic behaves the same even if another thread acquires the lock between leading execution and confirmation by the trailing thread.

The function `pthread_mutex_trylock` behaves like `pthread_mutex_lock`, but it does not spin. Instead, if the mutex is locked, the function first performs a context switch and then returns `EBUSY` after the execution switches back. Note that context switch should not occur at that time, as the thread should first unlock its other mutexes. However, the PARSEC benchmarks do not use the `pthread_mutex_trylock` function as intended, making the context switch necessary to avoid deadlocks.

The `pthread_mutex_getprioceiling` and `pthread_mutex_setprioceiling` functions are not implemented, as they are not used by the PARSEC benchmark suite.

8.3.3. Barrier

Our implementation provides reusable barriers. Each barrier stores the total number of threads and the number of times execution has arrived at the barrier. The `pthread_barrier_init` function sets the number of the threads to the passed parameter and the arrival count to zero. The `pthread_barrier_destroy` function only sets the number of threads to zero, as no dynamic memory needs to be freed. The `pthread_barrierattr_` function group is not implemented, as it is not used by the PARSEC benchmarks.

The `pthread_barrier_wait` increments the arrival count by one either atomically or with transactions. The arrival count before the addition is divided by the thread count to determine the generation. This generation is compared to the current arrival count divided by the thread count in a loop. Every 10,000 iterations a context switch is attempted. Once sufficiently many threads have arrived at the barrier the generation changes and the function is exited. As there is no cleanup necessary, other threads can already start waiting for the next generation before all threads of the current generation have left.

This implementation has a high performance impact in the redundant multiversioning variant. Every thread has to write the same cache line, which generates a new version for every thread. However, the maximum number of versions per cache line is limited. Therefore, further leading threads have to wait until they are confirmed by the trailing threads and can be freed. This causes all leading threads to wait longer than necessary, as they cannot leave the barrier before all leading threads have entered.

However, this implementation is still better than the common implementation using both an arrival and leave counter [34], as this would require two writes in separate transactions causing the bottleneck twice. A possible performance optimization is to split the counter into multiple parts, which are stored in separate cache lines. The threads choose the cache line by taking their ID modulo the part count. This alleviates the version requirement when writing, but creates an overhead when reading, as it is now necessary to sum all parts up.

8.3.4. Once

The function `pthread_once` uses a control structure to ensure that the passed function pointer is called exactly once and completed before `pthread_once` returns. The control structure consists of two booleans, one indicating that execution has started and one indicating that it has completed. First, the boolean indicating that execution has started is set to `true` using an atomic operation or a transaction respectively. If it was `false` before, the function pointer is called. Afterwards, the boolean indicating completion is set to `true` and the function `pthread_once` returns. However, if the boolean indicating that execution has started was already `true`, `pthread_once` waits for completion in a loop. Every 10,000 iterations a context switch is attempted. As soon as completion is signaled, the function returns. Note that the function can return without executing a single loop iteration if both booleans are already set to `true`.

8.3.5. Scheduling

Scheduling functions are not implemented, as they are not used by the PARSEC benchmarks. Internally, simple FIFO scheduling is used: All halted threads are stored in a queue. They enter the end of this queue when they are started or yield. When a core becomes ready, either because its current thread completes or yields, it fetches the first thread in that queue atomically. It then switches to this thread to continue execution. If the thread is still blocked, it might yield immediately, resulting in the next thread being fetched.

8.3.6. Condition Variable

Our implementation of condition variables is ticket-based. Each condition variable stores the number of dispensed tickets and the number of called tickets. Note that these are just integers. The function `pthread_cond_init` initializes them both to zero.

The function `pthread_cond_wait` draws a ticket by incrementing the dispensed tickets by one. This is performed with an atomic operation or a transaction respectively. The result of this summation is the ticket the function is waiting for. It then unlocks the mutex and waits until the ticket is called. The wait is performed by spinning. Every 10,000 iterations a context switch is attempted. When the number of called tickets is equal or greater than the drawn ticket, the mutex is locked again, which concludes the operation.

The function `pthread_cond_signal` wakes a single thread by incrementing the number of called tickets by one. However, if the number of called tickets is already equal to the number of dispensed tickets the operation is skipped. The function `pthread_cond_broadcast` wakes all waiting threads up by setting the number of called tickets to the number of dispensed tickets.

The function `pthread_cond_destroy` is empty, as no additional memory or system resources are allocated for our implementation of condition variables. The function `pthread_cond_timedwait` and the `pthread_condattr_` function group were not implemented, as they are not used by the PARSEC benchmarks.

8.3.7. Thread Local Storage

Thread local variables are stored in an array in `thread_info`. Currently, our implementation supports up to 256 thread local variables, which is sufficient for all benchmarks. The function `pthread_setspecific` determines the current thread by calling `pthread_self`. It then sets the corresponding array entry to the passed value. The function `pthread_getspecific` performs the same operation, but returns the entry instead. Note that `thread_local` keyword is not supported by the

MicroBlaze gcc. Therefore, explicit calls of the functions `pthread_setspecific` and `pthread_getspecific` are required on every access.

The function pointers to the destructors of all thread local variables are stored in a global array. These destructors are called when the thread is cleaned up, for example in the `pthread_join` function. As the array is not thread local, every destructor is only set once and applies for all threads, in which the corresponding thread local variable is not `NULL`. The function `pthread_key_create` determines the next free entry. The function pointer in the array is set to the passed destructor. The index is returned as key for usage in the `pthread_setspecific` and `pthread_getspecific` functions. The function `pthread_key_delete` is not implemented, as it is not used by the PARSEC benchmarks.

8.3.8. Other

Reader Writer Locks are not implemented, as they are not used by the PARSEC benchmarks.

Spin locks are not implemented, as they offer little advantage, since regular locks spin before yielding anyways. A potential spin lock implementation would be very similar to the mutex implementation, just without yields. However, on a platform without timer interrupts, it can be dangerous to never yield, as an application, which encounters a locked spin lock on a single core machine, will deadlock.

According to the specification [33], it is acceptable for the concurrency functions `pthread_getconcurrency` and `pthread_setconcurrency` to have no effect except storing the parameter value. However, as no PARSEC benchmark uses them, they were left unimplemented.

Cancellation is not implemented, as it is not used by the PARSEC benchmarks. This includes the `pthread_cleanup_` group of functions, which are also not implemented.

The functions `pthread_getcpuclockid` and `pthread_atfork` are not implemented, as they are not used by any PARSEC benchmark.

8.4. Atomic Operations

Many applications including some PARSEC benchmarks use atomic operations for synchronization. Therefore, it is necessary to handle them to execute the benchmarks on our platform.

8.4.1. Atomic Operations in Transactions

The semantic of atomic instructions in transactions cannot be defined easily. For example, one could expect the result of an atomic write to become visible to other cores immediately. However, writes inside of transactions are only visible after the transaction commits.

Delaying the atomic write until the transaction commits is risky, as the programmer might use it to create a critical section. If this critical section interacts with components, which cannot be rolled back, this might lead to an invalid state.

If atomic writes become visible immediately, but non-atomic writes do not, this can lead to inconsistent views of memory. For example, a thread might allocate an object and then publish its pointer using an atomic write. However, the other thread reading the pointer does not see the object yet. So this approach is not applicable to all applications, either.

In our experience, the best approach for user-level applications is to treat atomic instructions in transactions as regular memory accesses. User-level applications usually do not interact with components, which cannot be rolled back. Therefore, the risk that two threads enter a critical section at the same time is nonhazardous for them. However, the atomic writes becoming visible so delayed usually decreases performance significantly. We did not find any approach, which works for operating system kernels.

8.4.2. Conversion of Atomic Operations

As the semantic of atomic operations in transactions is unclear, it is advised to convert them to transactions. As transactions are atomic, it is sufficient to execute the equivalent regular code in a transaction. For example, the `atomic_fetch_add` operation atomically increments a memory location and returns the old value. The same can be done using a transaction as follows:

```
int atomic_fetch_add(int* obj, int arg)
{
    tm_begin();
    int res = *obj;
    *obj += arg;
    tm_commit();
    return res;
}
```

Listing 8.1: Implementation of atomic fetch and increment using transactions

Besides arithmetic atomic operations, atomic compare exchange is often used. This operation compares a memory location to a register value. If they are equal, the memory location is overwritten with another register value. Otherwise, the register

containing the expected value is overwritten with the actual value. A flag indicates which case occurred. These steps are all executed atomically, preventing any other cores from interrupting them. The same logic can be implemented using transactions as follows:

```
bool atomic_compare_exchange_strong(int* obj, int* expected, int desired)
{
    tm_begin();
    if (*obj == *expected) {
        *obj = desired;
        tm_commit();
        return true;
    } else {
        *expected = *obj;
        tm_commit();
        return false;
    }
}
```

Listing 8.2: Implementation of atomic compare and exchange using transactions

Some architectures differentiate between weak and strong atomic compare exchange. A weak atomic compare exchange is allowed to fail even if the values are equal, while a strong one must always succeed in this case. As the transaction guarantees atomicity, the transactional implementation shown here is strong. However, it would be possible to convert it to a weak one by setting `expected` to the memory value and always returning `false` when the first transaction aborts. This might offer a performance advantage in applications where the memory location changes often and is unlikely to change back to the initial expected value.

Note that an overuse of transactions to replace atomic operations can lead to many short transactions. This can affect performance, as every transaction start and commit has a small overhead. In this case, it is advantageous to combine multiple converted atomic operations in a single transaction. However, care has to be taken to ensure that the maximum transaction size is not exceeded. Additionally, the combination can raise abort rate if contention is high.

An example for atomic operations, which are best combined into a single transaction, can be found in the benchmark *canneal*:

```
inline void Swap(AtomicPtr<T> &X) {
    //define partial order in which to acquire elements to prevent deadlocks
    AtomicPtr<T> *first;
    AtomicPtr<T> *last;
    //always process elements from lower to higher memory addresses
    if(this < &X) {
        first = this;
        last = &X;
    } else {
        first = &X;
        last = this;
    }
}
```

```

    }

    //acquire and update elements in correct order
    T *valFirst = first->Checkout();
    T *valLast = last->PrivateSet(valFirst);
    first->Checkin(valLast);
}

```

Listing 8.3: Implementation of the `Swap` function of the `AtomicPtr` class in the benchmark *canneal* using atomics

```

inline void Swap(AtomicPtr<T> &X) {
    tm_begin();
    std::swap(*this, X);
    tm_commit();
}

```

Listing 8.4: Implementation of the `Swap` function of the `AtomicPtr` class in the benchmark *canneal* ported to transactions

Listing 8.3 shows the implementation of the `Swap` function using atomics. This function swaps two `AtomicPtr`s atomically. As there is no atomic instruction on the MicroBlaze, which can access two memory locations simultaneously, the first pointer is atomically copied to a local (register) variable and set to a reserved value in the function `Checkout`. This value signals every other thread that it has to wait for the operation to complete. The value of the first pointer (in the register) is then atomically swapped with the second pointer. Lastly, the first pointer is set to the old value of the second pointer, which is currently stored in a register. To avoid deadlocks, additional logic, which ensures that pointers are always processed in ascending order of the address of the pointer itself, is required.

Listing 8.4 shows how the logic is simplified, when transactions are available. This variant simply calls the `swap` function of the standard library inside of a transaction. The transaction ensures that the swap is executed atomically, even though the standard library function uses multiple memory accesses. No pointer sorting or reserved values are required. Note that this is an optional transformation to improve performance and simplicity. Instead, a direct replacement of the atomic operations is also possible.

8.4.3. Fallback for Uncaught Load Linked/Store Conditional

The only atomic instructions available on the MicroBlaze are load linked/store conditional. All other atomic operations are constructed using these instructions. If an application using transaction still has load linked/store conditional instructions left, it can still function, as hardware fallback is available. When a load linked instruction is executed in a transaction, the monitor is set to the corresponding address as if it was executed outside of a transaction. However, the address is not really monitored.

This is not necessary, as transaction semantics guarantee that the value is unchanged if the transaction commits. Instead, the monitor is reset when the transaction ends by either commit or abort. This ensures that the trailing core can correctly reproduce the first store conditional in a transaction. However, it can lead to unnecessary failed store conditionals. These unnecessary failed store conditionals can also occur on other platforms. For example, ARM recommends unconditionally resetting the monitor on context switch or exception [8]. Therefore, we expect applications to be able to handle a small number of spurious fails.

8.5. Benchmark Details

Even after providing the missing system calls and porting the pthreads library, most benchmarks did not run. Many did not even compile and updates of obsolete syntax were necessary. Some also contained race conditions, which would interfere with fault injection analysis. Therefore, we performed necessary changes, while trying to preserve the timing behavior of the benchmarks. Appendix B contains descriptions of the necessary changes grouped by benchmark.

8.6. Summary

We were able to run the PARSEC benchmark suite on our FPGA prototype without an operating system. A launcher enables us to start the benchmarks in batch. It also successfully validates their results against x86 execution. We had to implement several APIs for the benchmarks to run. This includes a memory file system. Other functions had to be wrapped to ensure thread safety. We implemented part of the pthreads library with both atomic operations and transactional memory. Therefore, we can determine a baseline and compare our approach against it. Our pthreads implementation supports thread, mutex, barrier, once, condition variable and thread local storage functionality. We have replaced all atomic operations in the PARSEC benchmarks with transactions. However, a general approach to execute atomic operations in transactions was not found.

Even with all required library functions available, most PARSEC benchmarks did not compile or run. Therefore, we had to modify them to remedy their problems. Common issues include obsolete constructs, which are not supported by the MicroBlaze gcc, and programming errors, which do not manifest on more sophisticated platforms due to their better error handling.

9

Multi-Threaded Evaluation

This chapter contains an evaluation of the multi-threaded fault tolerance approach, which was implemented on an FPGA. We evaluated the runtime and compared it to the variant without fault tolerance and two naive approaches to fault tolerance. The impact of several optimizations was also measured. The error detection latency was evaluated. Finally, we performed fault injection analysis. Major parts of this evaluation are also contained in [3].

Table of Contents

9.1	Methodology	110
9.2	Execution Time Overhead	111
9.2.1	General Results	111
9.2.2	Benchmark canneal	113
9.2.3	Benchmark streamcluster	114
9.2.4	Race Conditions	115
9.2.5	Optimizations	116
9.3	Error Detection Latency	119
9.4	Fault Injection	120
9.5	Summary	123

9.1. Methodology

We implemented our approach on the Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit. This board features the XCVU9P FPGA and two 4 GB DDR4 memories. A USB port is available for JTAG and UART. The other components on the board were not used for our evaluation.

Our design features 12 MicroBlaze cores with support for single precision floating point operations. The cores are connected to coherent private data caches and instruction caches (each 16 kB, 4-way set associative). The caches are interconnected to two memory controllers and a UART module. Each memory controller is connected to one memory module. All even cache lines are stored in one module and all odd cache lines in the other. We use custom caches to implement our extensions, which are addressed by a memory-mapped interface, as described in Section 7.1. Registers are backed up using the trace port. Thus, no changes to the closed-source MicroBlaze cores were necessary. The design runs at 50 MHz with the main limiting factors for the clock rate being the performance counters and assertions.

We used the PARSEC benchmark suite [15] version 3.0 for the evaluation. Note that this is not a throughput evaluation with multiple single-threaded processes, but a runtime evaluation, where the benchmarks are run with multiple synchronized threads. A detailed description, how the benchmarks were ported to our platform, can be found in Chapter 8. To support the benchmark execution on the MicroBlaze, we had to port the pthreads library. As the benchmark *freqmine* does not support pthreads execution, it is missing from the evaluation. The benchmarks *fluidanimate* and *facesim* are also missing, as they do not support arbitrary thread counts. In particular, 12 threads are not supported. Some other modifications were necessary to compile the benchmarks, as the MicroBlaze compiler does not support some old constructs and reserves additional keywords (see Appendix B). As far as possible, we avoided changes that can affect performance.

Except for fault injection, the benchmarks used the *simmedium* configuration. The limiting factors for the input set size are the slow transfer of the input files via JTAG and the large number of different configurations. In total, this evaluation requires 1456 individual benchmark runs. The benchmarks were executed entirely, but to measure the execution times only the region of interest was considered. To validate the correct execution, the outputs were copied back to the host machine and compared to x86 executions with the same thread count. We had to add additional outputs to the benchmark *raytrace*, as it discards its results.

9.2. Execution Time Overhead

We expect the execution time of a dual modular redundant approach to be between the runtime of the non-redundant variant with half the core count and twice the runtime of the non-redundant variant with the same core count. The execution with half the core count forms a lower bound, as our approach executes the benchmark twice (once on the leading cores and once on the trailing cores). However, this estimation is overly optimistic, as our approach requires continuous communication. Issues like false sharing, which reduce the scaling of the non-redundant multi-threaded application, also affect our approach negatively. In theory, it is possible for our approach to outperform this bound if the application blocks frequently due to synchronization, but still scales very well. However, we consider this kind of application as purely academic and do not expect it to occur in practice.

Executing the application twice one after another forms an upper bound for the runtime. Notice that this naive approach does suffer from two major disadvantages: The error detection latency is very long, as it lasts from the first instruction of the first run to the comparison after the execution of the second run. In addition, it is not possible to easily implement recovery, as executing the program a third time takes a long time and the initial state might not be available anymore.

9.2.1. General Results

For 12 cores, the geometric mean of the slowdown comparing the redundant variant to the non-redundant baseline is 2.16. We use the geometric mean ($\sqrt[n]{x_1 \cdot x_2 \cdots x_n}$), as it is the only correct average of normalized numbers [22]. Below, we describe the results shown in Figure 9.1 in detail.

The benchmark *vips* behaves as expected. The execution without redundancy follows Amdahl's Law. The execution time with redundancy is in the expected range (gray area in Figure 9.1). That is, it lies between the runtime of two parallel executions and two successive executions. For the benchmark *vips*, all these runtimes are very similar.

The benchmark *blackscholes* is embarrassingly parallel. Some of the used floating point operations are implemented in software on the MicroBlaze, which results in the threads having different runtimes. As there is no work balancing and due to the memory controller acting as a bottleneck, the benchmark does not reach a perfect speedup. There are minor false sharing issues with our approach, as the data is split over multiple arrays with no padding between threads. However, very little global data is written. Therefore, transaction conflicts are still rare and the impact of the issue is small.

The benchmark *bodytrack* uses a thread pool for parallelization. As this thread pool contains as many threads as cores and there is an additional main thread, frequent

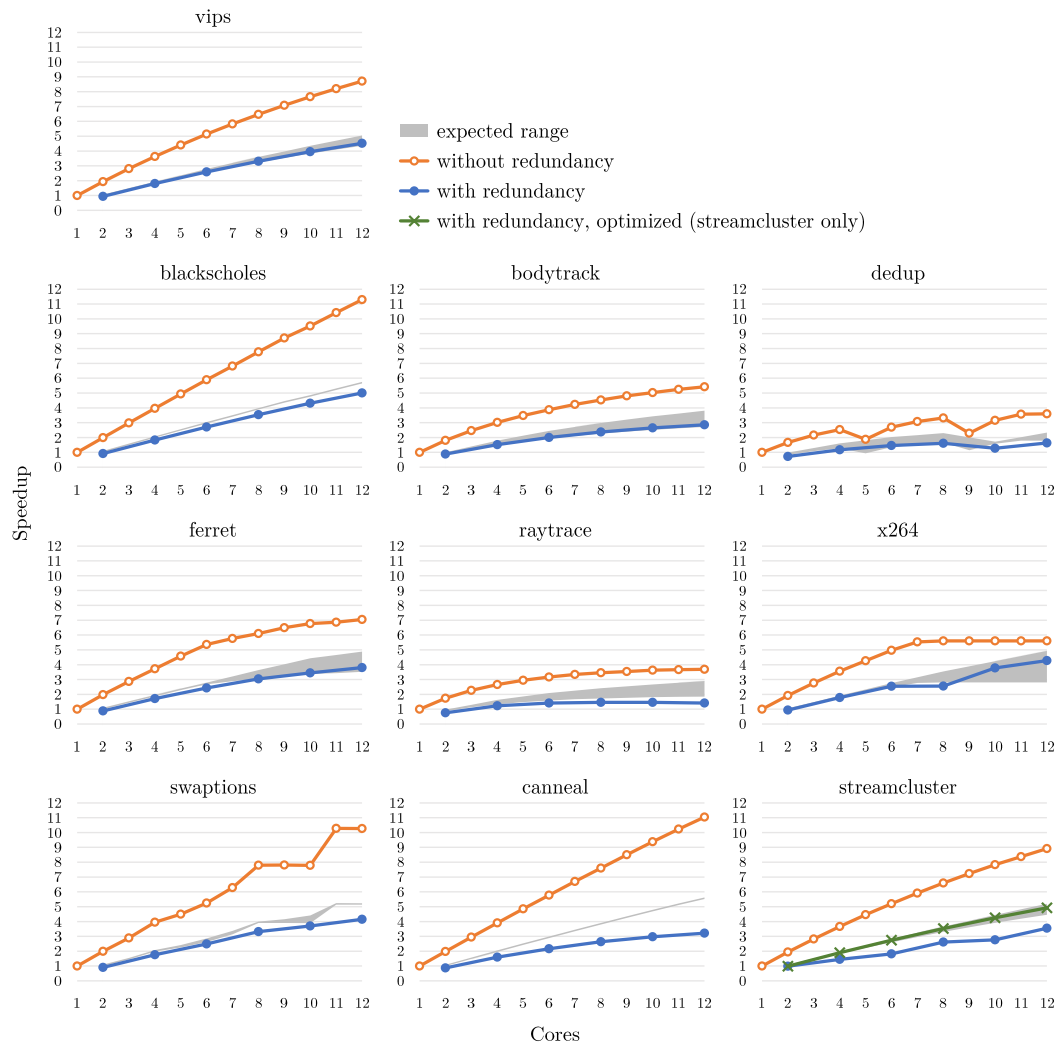


Figure 9.1.: These charts show the speedup of the various PARSEC benchmarks in the different configurations at a certain core count. All speedups are relative to the single-threaded execution without redundancy and consider only the region of interest. For the variant without redundancy, the benchmarks were launched with the core count as thread count parameter. For the variant with redundancy, half the core count (i.e. the leading core count) was used as parameter. The lower bound of the expected range is the execution without redundancy repeated twice. The upper bound of the expected range is the execution without redundancy executed twice in parallel with half the core count. Note that especially at low core counts the expected range is hidden behind the line for redundancy for some benchmarks, as the runtimes are so close together.

context switches are required. In addition, some parts of the application in the region of interest are not parallelized. These two aspects result in a shallow speedup curve. Our approach cannot take advantage of the lack of parallelization, as the sections are too long to cover them with the loose coupling. Thus, the speedup of our approach behaves more similar to the non-redundant variant, which executes one after the other, than the parallel one.

The benchmarks *dedup*, *ferret* and *raytrace* are limited by main memory bandwidth. A large shared L2 cache would most likely improve the performance for both the baseline and our approach. Our approach would profit further from it because the trailing cores access the same data as the leading cores with some delay. Thus, a sufficiently large shared L2 cache would completely eliminate the trailing memory accesses.

The benchmark *x264* stops scaling at high thread counts due to synchronization constructs. As the waiting threads do not consume any resources like memory bandwidth or trailing runtime, our approach performs well. The 8 core redundant run shows that executing more iterations for benchmarking would be favorable. Sometimes the benchmark *x264* takes longer to complete for no obvious reason. We suspect that an unlucky memory allocation ordering leads to an unfavorable interleaving of memory regions.

The benchmark *swaptions* shows irregular scaling. However, this is unrelated to the platform or the approach, as it is caused by the small input size. In the *simmedium* configuration, 32 work items are partitioned between the threads. This works out (relatively) well for e.g. 4, 8 and 11, but poorly for e.g. 9, 10 and 12, which results in the steps in the speedup graph. Note that there is one spare slot in the case of 11 threads. However, the impact of this single slot is minimal and the adjacent thread counts perform much worse.

The benchmarks *canneal* and *streamcluster* perform worse than the expected range. An explanation and proposed changes follow below.

9.2.2. Benchmark canneal

The benchmark *canneal* mostly uses a custom synchronization mechanism. Pointers are accessed automatically and can contain a special value to indicate that the object is locked for writing. If the object is locked, busy waiting is performed. This synchronization approach does not scale well with multiversioning, as the many explicit atomic operations result in many small transactions. In addition, the system cannot detect when a thread is waiting, which means the trailing cores waste much time to confirm waiting loops. To optimize such an application for our system, one would need to replace the atomic operations by transactions. In this case, this would be easily possible, as the main operation is a simple swap, which easily fits in a single transaction, eliminating the need for locking altogether. Section 8.4.2 demonstrates

this transformation. Note that regular transactional memory optimization rules still apply (i.e. those transactions would most likely be too small). Since we did not perform a comprehensive optimization of the benchmark and the simple transformation itself offers little performance benefit, Figure 9.1 shows only the variant with direct conversion of the atomic operations.

9.2.3. Benchmark *streamcluster*

The benchmark *streamcluster* uses many barriers. Sometimes barriers follow each other directly with no code between them. Barriers are problematic for our approach, as they tend to run out of free versions, which forces the leading cores to wait for the trailing cores to catch up. If the code executed between the barriers is too short, the transaction will not reach its intended length, leaving little time for the trailing core to confirm the barrier operation before waits become necessary. To optimize the benchmark, one should try to reduce the number of barriers needed. For our approach, it is better to execute short serial work (like adding the result of all threads) redundantly on all threads instead of just one, as this benchmark does, to remove additional barriers.

```
static double gl_cost_of_opening_x;
// ...
work_mem[pid*stride + K+1] = cost_of_opening_x;
pthread_barrier_wait(barrier);
if( pid==0 ) {
    gl_cost_of_opening_x = z;
    for( int p = 0; p < nproc; p++ ) {
        gl_cost_of_opening_x += work_mem[p*stride+K+1];
    }
}
pthread_barrier_wait(barrier);
if ( gl_cost_of_opening_x < 0 ) {
// ...
```

Listing 9.1: Selected code sample of the *streamcluster* benchmark

```
double gl_cost_of_opening_x;
// ...
work_mem[pid*stride + K+1] = cost_of_opening_x;
pthread_barrier_wait(barrier);
gl_cost_of_opening_x = z;
for( int p = 0; p < nproc; p++ ) {
    gl_cost_of_opening_x += work_mem[p*stride+K+1];
}
if ( gl_cost_of_opening_x < 0 ) {
// ...
```

Listing 9.2: Selected code sample of the *streamcluster* benchmark after optimization to reduce the number of barriers

Listing 9.1 shows a code section of the benchmark *streamcluster*. In this section, the first thread adds the local `cost_of_opening_x` (stored in `work_mem`) of all threads together and stores it in a static variable called `gl_cost_of_opening_x`. Two barriers are required to protect this code section: The barrier before the code section is necessary, as the threads calculate the local `cost_of_opening_x` in the previous section. The barrier after the code section is required, as the threads read `gl_cost_of_opening_x` in the following section. This implementation behaves poorly in redundant mode, as the section is very short and the trailing cores have most likely not finished validating the previous transaction, when it completes. Therefore, the additional versions for the barrier have not been freed yet and the leading cores have to wait for them to become available again before they can enter the barrier.

This section can be optimized by making `gl_cost_of_opening_x` a local variable as shown in Listing 9.2. All threads sum up the values individually. This makes the second barrier unnecessary. It does not introduce any new versioning overhead, as all threads only read the shared memory.

It can also be seen that the benchmark prefers even thread counts in the multiversioning variant. If all threads try to write to the same cache line at the same time, the available speculative versions (two in this implementation) for this cache line will run out quickly. Further threads then have to wait until those versions get confirmed by the corresponding trailing transactions. As this benchmark makes heavy use of barriers, threads will always reach such code sections at the same time, which means that it will be completed in batches of two. Thus, an odd thread count will result in another batch, which contains only one thread. Writing such code should be avoided, as there will also be some serialization when executed without redundancy due to the cache line bouncing between the cores. However, a cache miss is significantly cheaper than a trailing transaction, which makes the effect less prevalent for the baseline.

The benchmark *streamcluster* is also a prime example for the impact of false sharing. The source code contains a constant called `CACHE_LINE`, which controls the padding between the memory regions of the different threads. It is initially set to 32. However, the cache line size, which is used in our platform, is 64. Changing this value accelerates the application by 38.6%.

9.2.4. Race Conditions

Some benchmarks suffer from race conditions. This causes issues, when those benchmarks are executed redundantly, as it can no longer be guaranteed that the trailing core reads the same data as the leading core. The mismatch is then detected as a transient fault and all cores are rolled back in order to retry. Depending on the frequency of the race condition, it can immediately occur again.

The optimal solution is fixing the source code, as those race conditions can result in wrong results even on other architectures. If this is too difficult, it is also possible to enable conflict detection not just for explicit but also for automatic transactions. This enables the leading cores to detect the conflict, before it can affect the trailing cores. Thus, only a single leading core will roll back instead of the whole system. However, this does not resolve the race condition itself, meaning the application might still output wrong results. Though, it can reduce the frequency of the race condition, as the transactions produce a coarser interleaving. In addition, enabling the conflict detection for automatic transactions will increase the impact of false sharing.

We observed race conditions in two of the tested benchmarks. Enabling conflict detection for automatic transactions in those benchmarks results in an additional overhead of 4.1% for *x264* and 10.7% for *cannal*.

9.2.5. Optimizations

We have implemented several optimizations to improve performance. These optimizations are already enabled in Figure 9.1.

Figure 9.2 shows the effects of each hardware optimization by itself. We did not enhance the baseline pthreads library with support for redundant execution. Additionally, using a version of a library that is unsuitable for the employed hardware is unreasonable. Therefore, we did not evaluate it. For this evaluation, each configuration was executed three times. The median execution time was used to calculate the speedups shown in Figure 9.2.

The speedup of the different optimizations does not stack additively, as they aim at similar aspects. If one optimization has already reduced the number of cache misses, the possible further acceleration by other similar optimizations is limited. Furthermore, some optimizations directly reduce the effect of others. For example, not validating automatic transactions reduces the effect of the Bloom filter as less conflict detection is performed. Compression reduces the effect of fresh fetches, as fallback cache lines are required less often.

At the same time, it is also possible that the combination of two optimizations results in a larger acceleration than the sum of them. For example, a benchmark might be limited by the performance of its trailing cores. This makes all optimizations affecting the leading cores seem useless by themselves. Optimizations affecting the trailing cores only improve runtime to the level of the leading cores. However, if all optimizations are applied at once, the achieved acceleration is greater, as now neither the initial runtime of the leading cores nor the trailing cores is limiting.

Disabling the validation of automatic transactions has a large effect on some benchmarks. If the effect is that large, it cannot be explained by reduced conflict detection

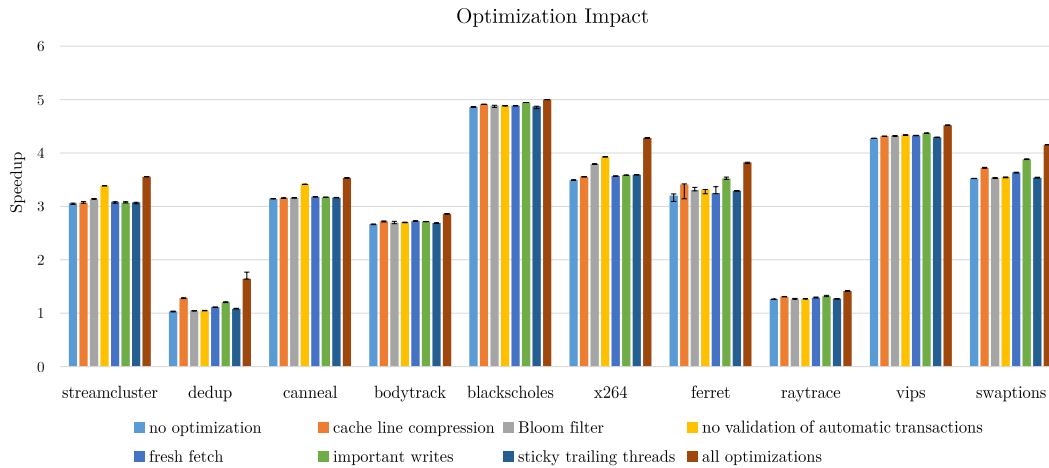


Figure 9.2.: This bar chart shows the speedups of various configurations relative to the single-threaded baseline without transactions or redundancy. The configurations were run on six leading and six trailing cores with multiversioning enabled. Only the region of interest is considered. Each bar represents the median runtime for that benchmark configuration. The error indicators extend to the slowest and fastest runtimes, respectively. The bars are in the same order as the legend entries.

overhead alone. This means that there are transactions, which aborted and retried if automatic transactions are validated. The obvious explanations for this behavior are false sharing and race conditions. The variant of the benchmark *streamcluster*, which is run here, uses the wrong cache line size (see Section 9.2.3). Therefore, this benchmark suffers from false sharing.

However, the benchmarks *canneal* and *x264* are not free from race conditions. Disabling the validation of automatic transactions for those benchmarks is not without downsides. It can no longer be guaranteed that the trailing core reads the same data as the leading core. The mismatch is then detected as a transient fault and all cores are rolled back in order to retry. Depending on the frequency of the race condition, it can immediately occur again. In this evaluation, there was an average of 20.5 global rollbacks in *canneal* and 0.8 in *x264* when the validation of automatic transactions is disabled. The optimal solution would be to fix the source code, as those race conditions can result in wrong results even on other architectures.

Other effective optimizations include cache line compression and important writes. We did not observe any benchmark becoming slower if a certain optimization is enabled. Additionally, the hardware cost for most optimizations is negligible. Some like fresh fetch are probably already included in more sophisticated systems. Therefore, there is no downside to enabling most optimizations even if the achieved acceleration is low. Disabling the validation of automatic transactions is the obvious exception because it can lead to livelocks due to infinite global rollbacks as described above.

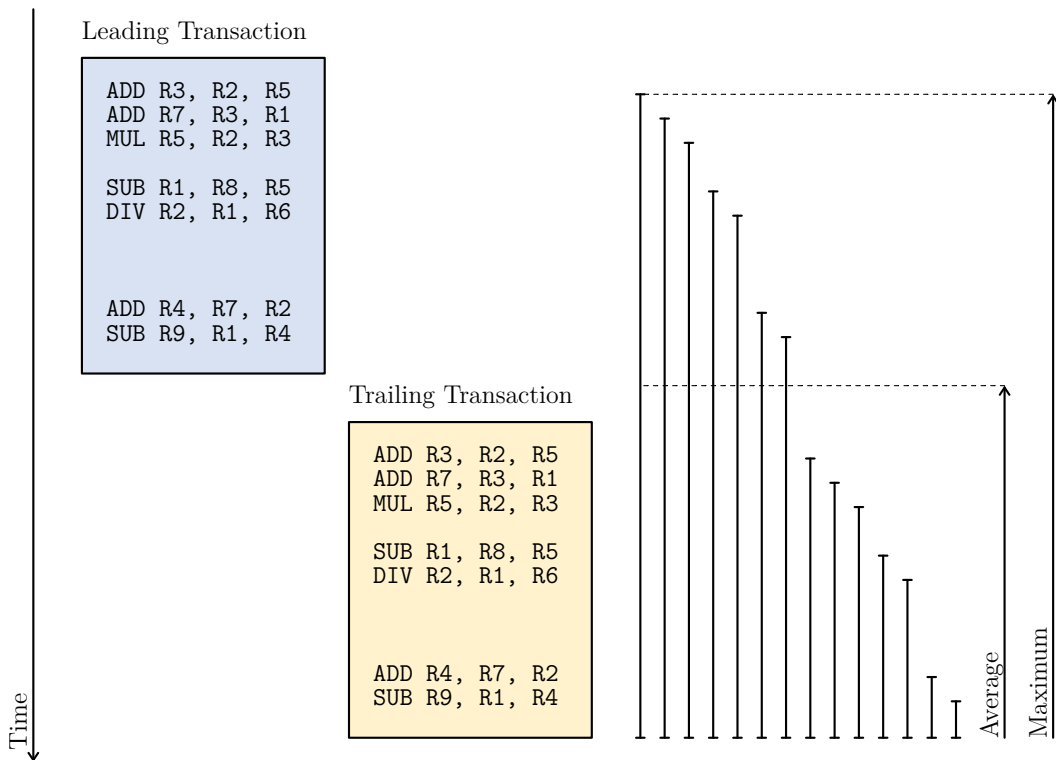


Figure 9.3.: The calculation of the error detection latency is based on instructions and transaction commits. Every leading transaction (blue) is examined together with its corresponding trailing transaction (yellow). The time span between every instruction and the commit of the trailing transaction is calculated. Cycles, in which no instruction was completed, are ignored. The maximum latency is the longest of these time spans, i.e. from the first instruction to the commit of the trailing transaction. The average latency is the average of all these time spans. Note that for the same execution of the same transactions the maximum latency is roughly double the average latency due to the measurement procedure. To aggregate the values for the whole benchmark execution, the weighted average and the maximum are used respectively.

Benchmark	Average [cycles]	Maximum [cycles]
vips	9,833	181,129
blackscholes	9,985	25,263
bodytrack	9,855	30,976
dedup	8,621	150,333
ferret	9,697	156,534
raytrace	8,359	59,790
x264	10,492	108,198
swaptions	9,852	29,083
canneal	7,765	31,780
streamcluster	8,890	49,056
overall	9,335	181,129

Table 9.1.: The average error detection latency is the average number of cycles between every instruction and its corresponding checksum comparison. The maximum latency spans from the first cycle in a leading transaction to the checksum comparison of the corresponding trailing transaction. These values were measured for the whole benchmark with 6 leading cores and 6 trailing cores.

9.3. Error Detection Latency

We have also analyzed the error detection latency. The average and maximum values are shown in Table 9.1. Figure 9.3 shows the way average and maximum latency were calculated.

The resulting average values clearly reflect the targeted transaction duration of 10,000 cycles. Many automatic transactions hit this target quite accurately and a trailing core is ready right away to validate the transaction. However, for most benchmarks the average is lower, as synchronization operations explicitly commit the transaction before the time limit is reached. Some automatic transactions are longer than the target. This happens, as we can only commit transactions at memory instructions. We suffer from this constraint, because the MicroBlaze is closed-source. In a more comprehensive implementation, this would most likely not be an issue. If a benchmark makes heavy use of software floating point, this constraint can have a significant impact, as the gcc software floating point library avoids memory operations whenever possible and some operations take relatively long.

The worst-case error detection latency is significantly higher for most benchmarks, as the speculative nature of transactional memory can result in load spikes on the trailing cores. Those load spikes result in waiting times before a transaction can be validated. Another reason for large error detection latency are threads that switch from one trailing core to another and incur more cache misses than the corresponding leading transaction. Thus, the trailing core takes longer than the planned 10,000 cycles to complete validation. These extreme cases occur very rarely, though, which

makes it very likely that an error will occur during a time when the error detection latency is short. Note that there already is a two-fold increase between the maximum and average latency, as, for each transaction, the maximum latency spans from first instruction to the checksum comparison, while the average is taken from the latency between each instruction and the checksum comparison (see Figure 9.3).

If the error detection latency is too high for the intended application, it can be lowered by reducing the targeted transaction length. This does not only reduce the average, but also the maximum. One has to expect a decline in performance, though, as this will cause higher transaction boundary overhead overall. It can be considered to increase the targeted transaction length to reduce overhead. However, this will only work for some benchmarks. If the error detection latency becomes too large, cache lines will be evicted, before the trailing cores have validated them, which results in more cache misses. Thus, increasing the targeted transaction length will only improve performance for benchmarks with a low cache miss rate.

9.4. Fault Injection

To ensure proper operation of our system, we performed a fault injection analysis. The injection is performed in hardware, but is triggered by software running on the host machine. As the cores are closed source, we had to add the injection logic at gate level. We are also limited in the ways, in which we can inject faults. The injection is implemented by XOR gates in front of the register set's write port. The other input is set to zero by default, but the host machine can change this value. If a bit is set to one, it remains high until a register is written. This is done to increase the number of faults, which have a chance to affect the application output. Otherwise, many attempts would be wasted to cache misses or instructions, which do not write registers.

We collected the region of interest's start and end times for each benchmark. Before the benchmark starts, the host machine selects a random time in this period. It then triggers the injection once the selected time has elapsed. The core and affected bit are also chosen randomly. The only condition for a core to be eligible is that it has to be active. Therefore, the injection includes non-redundant, leading and trailing cores.

We have performed 50 iterations per benchmark for each the variant without and with redundancy. We used all 12 available cores and the maximum thread count. The validation of automatic transactions was enabled to prevent interference from unidentified race conditions. The benchmark *raytrace* was not included in the analysis, as it takes a long time (~ 1 h) for initialization. The benchmark *canneal* was excluded, too, as its output fluctuates too much to detect errors. Except for the benchmark *x264*, all benchmarks use the *simsall* configuration for quick injection

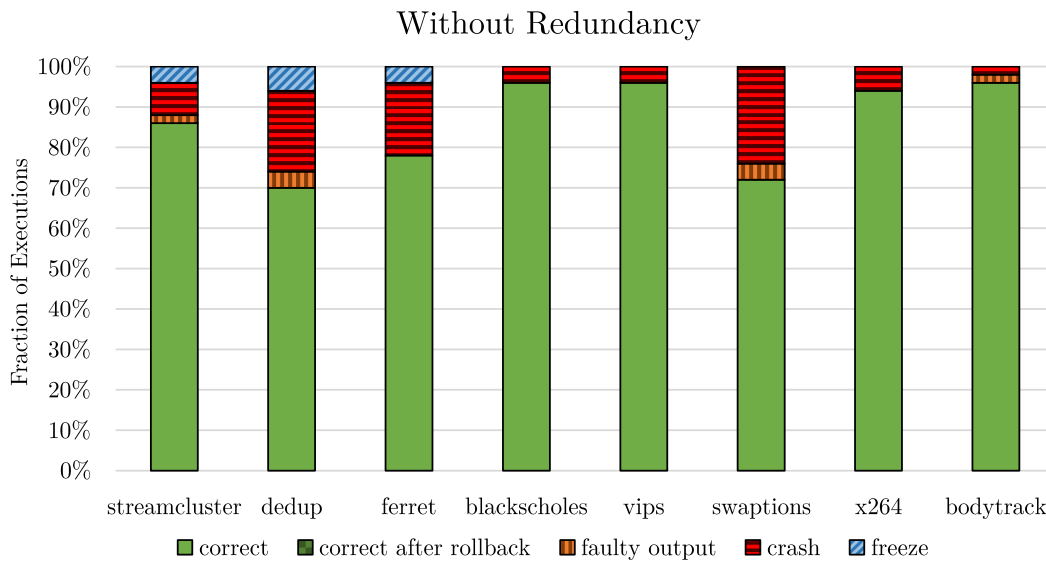


Figure 9.4.: Without redundancy, the effect of injected faults can be observed. Each benchmark was executed 50 times and a single bit-flip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. Rollbacks can only occur in the redundant variant with multiversioning.

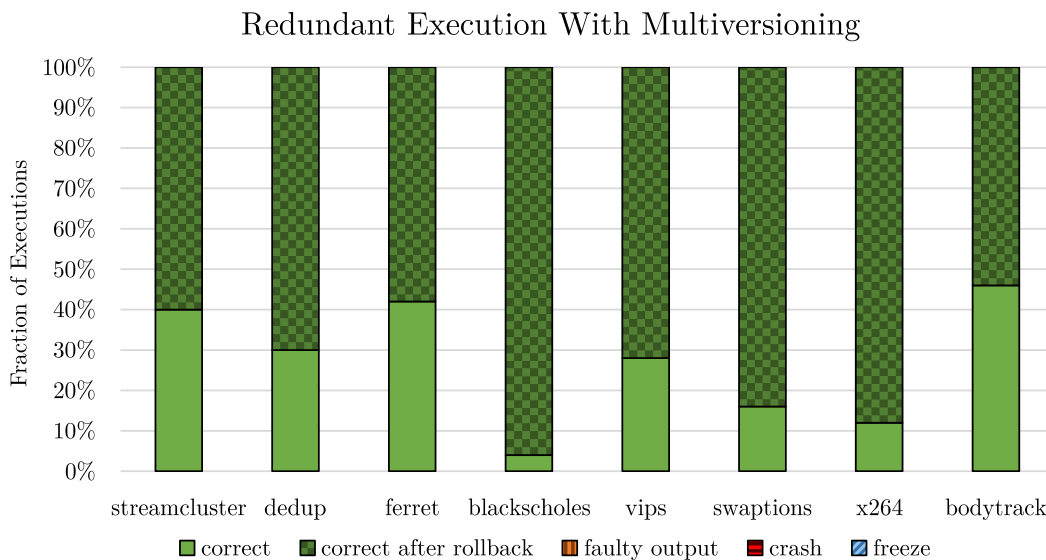


Figure 9.5.: This bar chart shows the results of the fault injection, when redundancy with multiversioning is used. Each benchmark was executed 50 times and a single bit-flip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. In this evaluation, every output was correct. Sometimes, a global rollback was required to achieve this.

attempts. The benchmark *x264* crashes if the number of threads is greater than the number of frames. Therefore, it requires the *simmedium* configuration.

We can validate the outputs of all used benchmarks. Therefore, we can differentiate the following groups of outcomes:

- Valid output, benign error, error corrected
- Invalid output, silent data corruption
- Freeze
- Crash

Furthermore, we can detect whether a global rollback was performed in the fault tolerant variant. Remember that a global rollback can only be caused by a checksum mismatch and the first checksum mismatch always causes a global rollback, but additional mismatches before the rollback do not cause another one. Therefore, the presence of the global rollback indicates whether the fault was detected.

Figure 9.4 shows the results of the fault injection without redundancy. Most runs complete correctly even if faults are present. Especially, the benchmarks *blacksholes*, *vips*, *x264* and *bodytrack* mask faults effectively. In the cases in which the bit-flip influences the output, the result is likely a crash. These crashes are mostly invalid instruction or unaligned access exceptions. An invalid instruction exception can be caused by a corrupted return address. Since no MMU is used, the core tries to execute the data at the invalid address as instructions instead of throwing a segmentation fault. However, it is very unlikely that a random memory location contains a sufficiently long sequence of valid instructions. The MicroBlaze cores can only handle aligned memory accesses. Therefore, an unaligned access exception is thrown if the lower bits of a pointer are corrupted. The benchmarks *streamcluster*, *dedup* and *ferret* also tend to freeze. This can happen if the most significant bit of a loop counter is flipped. As most loops use signed counters, this means that the loop will be executed around 2 billion times, which takes so long that the timeout in our benchmarking script triggers. Faulty outputs are possible if the flipped bit affects actual data. However, they are quite unlikely. In our evaluation, faulty outputs have only occurred in the benchmarks *streamcluster*, *dedup*, *swaptions* and *bodytrack*.

As shown in Figure 9.5, our multiversioning approach prevented all undesirable outcomes in our evaluation. In 72.8% of the runs, a global rollback was necessary. A global rollback only occurs if a checksum mismatch was detected. We did not observe any spurious or duplicate rollbacks, which indicates that fault detection and isolation work properly. The main cause why a bit-flip does not result in a checksum mismatch is that the containing transaction was aborted. Aborted transactions are not validated by the trailing cores, as they cannot influence the program output. Many aborted transactions happen while the thread is waiting for a synchronization construct or the core is idle. These bit-flips would not have affected program output

in the baseline variant either. The bit-flip itself can also be the reason for the transaction abort. For example, it can shift the stack pointer to overlap with a different thread. The conflicting memory accesses are detected and one of the transactions is aborted.

Note that a checksum mismatch does not necessarily mean that an undesirable outcome would have happened without redundancy. For example, it might hit a comparison, for which only the sign bit is relevant. In this case, bit-flips in the lower bits do not influence the program output. Therefore, the likelihood of a checksum mismatch correlates more strongly with the transaction success rate than an application's susceptibility to errors. For example, the benchmark *dedup* has the most undesirable outcomes in the baseline, but has executed slightly below average global rollbacks. Contrary, the benchmark *blackscholes* has executed the most global rollbacks, but is one of the benchmarks with the fewest undesirable outcomes in the baseline. The high number of checksum mismatches in this benchmark is mainly caused by the low number of transaction aborts, as the threads process independent work items.

The timing impact of global rollback was below the regular runtime fluctuation caused by varying memory response times and thread ordering. The largest contribution to the timing overhead is the staggered restart of the cores. Staggered restarts have the advantage that checksum mismatches caused by false sharing are resolved, as the involved transactions run successively. Therefore, an infinite loop of global rollbacks is avoided.

Currently, our prototype is limited in the types of faults it can handle. For example, injecting a large number of bit-flips will eventually lead to a freeze, as the core tries to access an invalid address. As it never receives a response, it cannot enter the interrupt handler to perform the rollback. Note that this situation is rare, as few address ranges behave this way. We plan to add an MMU to prevent such situations. Furthermore, we want to switch the recovery mechanism from interrupts to resets so that we can recover from more situations.

9.5. Summary

This evaluation shows that most benchmarks already perform well without changes to their source code. The slowdown (2.16 geometric mean) is in the range, which we expect for an approach, which executes the application twice. Considering that our approach also offers recovery, which usually requires three instances, this is a good result. If an application scales well without redundancy, good scaling can also be observed with our approach. Simple changes, like ensuring proper padding can result in large performance gains, e.g. 38.6% in *streamcluster*.

The impact of several hardware optimizations was evaluated. Disabling the validation of automatic transactions and cache line compression are the most impactful.

The approach features a low error detection latency of on average 9,335 cycles, making it suitable for use in systems, which require frequent output. Therefore, our approach should be applicable to most shared memory applications on general-purpose and embedded systems.

The evaluation has also shown that our approach can reliably detect errors and recover from them. A measurable performance overhead of the global rollback was not observed.

10

Related Work

A wide variety of fault tolerance approaches have been suggested in literature. Implementations in both software and hardware have been considered. Particularly, some approaches use HTM for recovery. However, the approaches in literature differ in significant ways from the approaches proposed in this thesis.

This chapter contains a short description of selected fault tolerance approaches. These are then compared in regards to various aspects like the support of multi-threaded applications or the availability of an integrated recovery mechanism.

Table of Contents

10.1 Hardware-Based Approaches	126
10.1.1 Lockstep	126
10.1.2 TMR: Triple Modular Redundancy	126
10.1.3 Diva: Dynamic Implementation Verification Architecture	127
10.1.4 AR-SMT: Active-stream/Redundant-stream Simultaneous Multi-threading	127
10.1.5 Slipstream Processor	128
10.1.6 ReStore: Symptom-Based Soft Error Detection in Microprocessors	128
10.1.7 Transient fault detection via simultaneous multithreading	129
10.1.8 SRTR: Simultaneously and Redundantly Threaded processors with Recovery	129
10.1.9 FaultM-multi	130
10.2 Software-Based Approaches	131
10.2.1 PLR: Process-Level Redundancy	131
10.2.2 SWIFT: Software Implemented Fault Tolerance	132
10.2.3 HAFT: Hardware-Assisted Fault Tolerance	132

10.2.4 COTS Fault Tolerance with Intel TSX	133
10.3 Comparison	134
10.4 Summary	136

10.1. Hardware-Based Approaches

Hardware-based approaches require additional hardware components dedicated to fault tolerance.

10.1.1. Lockstep

In lockstep execution, pairs of identical cores are used, which execute the same instructions [44, p. 212]. A fixed delay between the cores can be used to prevent common mode faults [9, p. 11]. The state of the cores is compared after each cycle and a fault is signaled if they do not match [44, p. 212]. However, recovery is not possible, as it is not known in which core the fault has occurred [6]. Due to the fixed delay and comparison latency, it is also no longer possible to flush the pipeline. Contrary, our approaches feature recovery. They also do not suffer from the tight coupling of lockstep processors, which makes performance optimizations and use on heterogeneous systems possible.

COTS (Commercial Of The Shelf) lockstep systems like the Arm Cortex-A78AE [5] offer a choice between a lockstep and a performance configuration, where both cores are available separately. Multi-cores with more than one lockstep pair are also possible. [7]

10.1.2. TMR: Triple Modular Redundancy

TMR (Triple Modular Redundancy) [58, pp. 19-22] solves the recovery issue of lockstep by adding a third core to each redundant group. If a fault occurs, the system can now perform a majority vote to determine the correct state. The faulty core can then be recovered by copying the state of a valid core. However, requiring a third duplicate core increases the overhead further. Contrary, our approaches suffice with two physical cores per logical core.

TMR systems are typically used in applications like avionics control systems, where every failure is critical. Therefore, common TMR implementations cannot be split into three separate cores. Contrary, our approach can also be used for mixed critical applications, which run low-criticality tasks on all cores without redundancy.

10.1.3. Diva: Dynamic Implementation Verification Architecture

The *DIVA (Dynamic Implementation Verification Architecture) Checker* [12] resides in additional pipeline stages and verifies the functional correctness of all instructions before they are allowed to commit. The *core processor* can continue execution speculatively while an instruction passes through the *DIVA Checker*. If the output of the *core processor* is incorrect, the pipeline is flushed to restore the state before the faulty instruction. The output of the *DIVA Checker* for this instruction then replaces the output of the *core processor*. Therefore, not only transient errors can be corrected, but also permanent errors in the form of design mistakes in the *core processor*. However, it is important that the correctness of the *DIVA Checker* is ensured. Otherwise, new errors can be introduced, as the correct output of the *core processor* is always replaced in case of a mismatch. However, the *DIVA Checker* is less complex than the *core processor*, which makes its validation easier.

Our approaches do not require the design of a separate checker core. If a smaller core already exists, we can use it for redundancy, as our approaches support heterogeneous multi-cores. When a mismatch occurs, we repeat the corresponding section. Therefore, we can detect and correct transient faults in both cores. However, we cannot correct design mistakes.

10.1.4. AR-SMT: Active-stream/Redundant-stream Simultaneous Multithreading

AR-SMT (Active-stream/Redundant-stream Simultaneous Multithreading) [53] runs two instances of a program concurrently on a SMT (Simultaneous Multi-Threading) processor. The first instance is called *active stream* and runs ahead. The *active stream's* results are pushed into a FIFO queue called *Delay Buffer*. The *redundant instruction stream* follows with a small delay and validates these results. This validation can happen before the corresponding instruction is committed, enabling rollback in case of an error.

Eric Rotenberg, author of [53], suggests the use of a trace processor [54]. On such a trace processor, different processing elements can be used for the execution of the corresponding trace in the *active stream* and the *redundant instruction stream*. Therefore, permanent faults can be detected reliably.

The approach requires twice the memory, as it runs two separate instances of the program. It cannot handle differences in memory, which can be caused by different execution schedules. Therefore, the approach is unsuitable for multi-threaded applications. Contrary, our approaches allocate storage in main memory only once. Our multiversioning approach can also handle multi-threaded applications.

10.1.5. Slipstream Processor

A *Slipstream processor* [50, 60] executes the same application twice in the form of an *advanced stream* and a *redundant stream*. The streams can be run on the same core on SMT processors or on different cores. Each stream has its own context and memory region. The *advanced stream* skips all instructions, which are predicted to be ineffectual. Therefore, it runs faster than the unmodified application. However, it is necessary to confirm the correct execution despite skipped instruction. This task is performed by the *redundant stream*, which executes all instructions. The *redundant stream* can use the outcomes of the *advanced stream* as predictions. Therefore, it is also accelerated.

The approach can detect faults by comparing the outcomes of the instructions, which are executed in both streams. However, not all faults can be detected and sometimes faults are indistinguishable from false predictions. A *recovery controller* is used to recover from detected faults.

Contrary, in our approaches false predictions do not affect reliability. However, this comes at the cost that the leading core is not accelerated. Additionally, the *Slipstream processor* does not contain mechanisms for input duplication. This makes it unsuitable for the execution of multi-threaded applications. Whereas, our multiver-sioning approach can handle execute multi-threaded applications.

10.1.6. ReStore: Symptom-Based Soft Error Detection in Microprocessors

ReStore [64] is a fault detection mechanism, which is purely based on the symptoms of an error. Such symptoms include exceptions, control flow misspeculations, cache and translation lock aside buffer misses. These symptoms do not cover all errors. For example, a bit-flip affecting only a data value and not a pointer or condition can remain undetected. However, *ReStore* still significantly reduces the FIT rate. Combining *ReStore* with additional protection for the most vulnerable parts of the processor, can increase error coverage further. We expect our approaches to perform better in concern to the detection of SDC, as we do not rely on symptoms, but validate every instruction.

ReStore creates checkpoints periodically using similar mechanisms like branch prediction. When a symptom detector triggers, rollback is performed and the affected section is repeated. The outcomes of the original and repeated execution are compared to detect whether a fault has really occurred or the symptom was part of regular execution. If the outcomes differ, a third execution can be performed to determine the correct outcome by majority vote. Unnecessary rollbacks caused by symptoms during fault-free execution cost some performance.

10.1.7. Transient fault detection via simultaneous multithreading

SRT (Simultaneous and Redundantly Threaded) processors [51] apply the concept of SMT to fault tolerance. The same application runs twice on the same processor. Its resources are shared between the two redundant instances similarly to a SMT processor. Both instances share the same memory. To ensure that they read the same values, either a *Load Address Buffer* or a *Load Value Queue* can be used.

SRT's sphere of replication can either include or exclude the register set. If the register set is excluded, register values need to be compared before writeback. In both variants, it is necessary to compare stores. This can be realized by separating the instances' pointers to the core's store buffer. At first, the store buffer entry is initialized with the leading instance's address and value. When the trailing instance's pointer reaches this entry, it is verified and cleared for transmission to the data cache.

The slack between the copies is regulated by varying their fetch priorities. This enables the leading instance to prefetch for the trailing instance. Furthermore, the branch outcomes of the leading instance are used as predictions by the trailing instance, which leads to a perfect branch prediction if no fault occurs. Our approaches also use the leading thread to prefetch and improve branch prediction for the trailing thread. However, we run the redundant copies on separate cores. This enables us to profit from heterogeneous multi-cores, in which the trailing cores can be more energy-efficient.

SRT does not feature an integrated rollback mechanism. Instead, it relies on separate recovery schemes. Contrary, our approaches can utilize the rollback capability of the HTM for a cheap and quick recovery.

The approach is extended in later work [43]. *Preferential space redundancy* instructs the core to use different components for leading and trailing execution, whenever this is possible. This improves the detection of permanent faults with little cost to performance.

CRT (Chip-level Redundant Threading) [43] is similar to *SRT*, but uses separate cores for the two redundant instances. If the processor supports SMT, a pair of cores can execute two threads redundantly. Each core executes one leading and one trailing instance, which permits both cores to profit from the acceleration of the trailing instance.

10.1.8. SRTR: Simultaneously and Redundantly Threaded processors with Recovery

The approach *SRTR (Simultaneously and Redundantly Threaded processors with Recovery)* [63] extends *SRT* to include recovery. The recovery mechanism uses the

processor’s rollback capability for speculative execution. Therefore, instructions may only commit after they have been confirmed. To avoid exhaustion of the speculative resources *SRTR* uses a shorter slack than *SRT*. As faults have to be detected early enough for recovery, register outputs have to be compared. A *register value queue* is used for this comparison to reduce pressure on the register file. *Dependence-based checking elision* is used to reduce the required bandwidth. This optimization avoids verification of registers if their value is used in other verified instructions, as a fault will propagate to the other instruction, which will cause its detection, when that instruction’s output is validated. However, the chains formed this way cannot be too long, as timely verification is required to avoid delaying the first instruction’s commit. It can also only be used with instructions, which do not mask faults, as the register storing the first instruction’s output could also be used in a later instruction, which is not yet visible to the core.

In our approaches, a checksum over stores and final register values is calculated. Therefore, only little data need to be transferred between the cores. It is also not necessary to correlate individual instructions for verification.

One of the main challenges of *SRTR* is that it has to work with speculative leading outputs, while *SRT* can use committed leading outputs. Therefore, *SRTR* can only forward branch predictions and cannot rely on the order of memory operations. Instead, *SRTR* tries to maintain an identical order of instruction in the threads’ active list. The position of an instruction in the active list can then be used to correlate them between leading and trailing execution. In our approaches, we can forward final branch outcomes and memory prefetches, as we do not depend on a small slack. We calculate the checksum at instruction commit. Therefore, it is not necessary to correlate individual instructions, which makes us independent of the concrete pipeline implementation.

10.1.9. **FaultM-multi**

FaultM-multi [85] extends *FaultM* [83, 84] with support for multi-threaded applications. Applications based on both atomic operations as well as transactional memory are supported. *FaultM-multi* is implemented in hardware and can recover from errors. It splits the execution in transactions called *rel-tx*. These transactions are executed redundantly on separate cores at the same time. At commit, their write-sets and register values are compared. The changes are written back to memory by only one of the cores and only if they match. Otherwise, both cores abort and repeat the transaction.

Atomic operations are executed as transactions, which only contain a single instruction. This ensures that they are immediately visible to other threads. No input duplication is used. Instead, the transactional memory’s conflict detection is used to

differentiate between error and competing write if a mismatch between the *rel-tx* occurs. If the application uses transactions for synchronization, the boundaries of the *rel-tx* are configured to match the transaction's boundaries. Therefore, no additional checkpointing or conflict detection mechanism is required.

The *FaultTM-multi* approach differs from ours in that the redundant transactions execute simultaneously. If one *rel-tx* finishes before the other, it has to wait. Additionally, exactly two cores have to be reserved for each *rel-tx* pair, making unequal counts of original and backup cores impossible. Our approaches do not suffer from these limitations. Additionally, we can use information like branch outcomes from the first transaction to accelerate the second transaction. This is especially useful on heterogenous systems. If *FaultTM-multi* was enabled on two cores with different speeds, it would be limited to the speed of the slower one.

10.2. Software-Based Approaches

Software-based fault tolerance approaches can run on regular hardware. However, some approaches require certain components like an HTM.

10.2.1. PLR: Process-Level Redundancy

PLR (Process-Level Redundancy) [55, 56, 57] is a software fault tolerance approach. It creates a number of redundant processes all executing the same binary. For interaction with the remaining system, the additional *figurehead* and *monitor* processes are also created, but they do not perform any actual computations. Correctness is ensured by comparing the redundant processes' outputs. A *system call emulation unit* performs this comparison, when a system call is invoked, and also duplicates the result of the system call. If at least three processes are used, recovery from errors is possible by terminating all processes, which disagree with the majority, and restarting them by cloning a correct process. Recovery with only two processes requires an additional checkpointing solution not provided by *PLR*.

PLR requires applications to be deterministic, as state has to be identical at each system call. However, in multi-threaded applications the memory state can differ depending on execution order, even if it is designed to produce a determined final state. Therefore, multi-threaded applications are not supported. Contrary, our multiversioning approach can handle the indeterminism introduced by different execution schedules. As *PLR* only detects mismatches at system calls, the error detection latency can be quite long. Additionally, some errors might not directly lead to a different system call parameter. In a high radiation environment, another error can occur before the first is detected. If *PLR* detects this error, it might copy the incorrect state resulting from the first error to the newly created redundant process. Therefore, the erroneous state might later be considered correct. Our approaches

feature shorter error detection latencies and can also handle environments with a high error rate.

PLR uses multiple processes with separate address spaces, which map to different physical memory locations. The additional checkpointing solution, which is required for rollback, or third redundant instance requires further memory. Contrary, our approaches use a single address space for both redundant instances. As the HTM's rollback capability is used, checkpoints are mostly stored in the caches and only little additional main memory is required.

10.2.2. SWIFT: Software Implemented Fault Tolerance

SWIFT (*Software Implemented Fault Tolerance*) [52] duplicates instructions during a compiler pass to provide fault tolerance. The sets of redundant instructions use separate registers, but share the same memory region, which is protected by an ECC. Therefore, stores are not duplicated and a register comparison is inserted before the store instead. A signature-based control flow checking approach is used to detect bit-flips in the PC or comparison flags.

The approach can only detect errors, but it cannot recover from them without an additional checkpointing mechanism. Contrary, our approaches use the rollback capability of HTM for recovery. In *SWIFT*, the memory's content is not allowed to change between redundant loads. Otherwise, the redundant registers will differ and a false fault will be detected at a later point. If the compiler knows that a memory value might change between loads, e.g. because it is accessed by an atomic operation in a multi-threaded program, it can avoid duplicating that load. Instead, the value is copied to the redundant register. However, this approach does no longer protect load instructions from bit-flips. The paper did not test any multi-threaded applications to confirm that the approach actually works with them. Our multiversioning approach, on the other hand, protects loads, even if they are part of a synchronization construct, without suffering from false faults. We evaluated the PARSEC benchmark suite, which confirms that multi-threaded applications run flawlessly.

10.2.3. HAFT: Hardware-Assisted Fault Tolerance

HAFT (*Hardware-Assisted Fault Tolerance*) [39] is a software fault tolerance approach for multi-threaded applications. It implements an extension to the LLVM compiler, which duplicates every instruction. The output of the duplicates is compared to detect errors. Additionally, the execution is split in transactions that are aborted if an error is detected to recover from it. As *HAFT* is implemented in software, the compiler has to insert appropriate instructions to start and commit these transactions. Due to the used HTM, Intel TSX, it is not possible to guarantee that

these static sections always fit in a transaction. In this case, the section will be run in a fallback mode without transactions.

Our approaches use dedicated hardware to split the execution into transactions. Therefore, we do not require an additional compiler step to insert transaction boundaries. Additionally, we can guarantee that every instruction is executed in a transaction.

*HAF*T assumes freedom from race conditions. Otherwise, additional transaction aborts occur. If a section is run without transaction, the approach cannot recover from an error occurring within it. If a race condition occurs between two redundant loads, it detects a DUE and crashes. However, the application without fault tolerance might not have crashed in this instance. As our approaches automatically sizes transactions in hardware, it cannot get into such a situation.

In *HAF*T, the redundant instructions utilize superscalar resources, which are unused in the non-redundant variant due to data dependencies. However, if there are no unused superscalar resources, execution times rise significantly, reaching up to 4 times the original execution time. Therefore, the approach is not viable on simpler cores. Contrary, our approach does not require superscalar resources, as it uses other cores for redundancy. Additionally, the total overhead is lower in our approach, since the comparison is implemented in hardware and does not consume execution units as in *HAF*T.

*HAF*T supports atomic operations, but cannot handle explicit transactions for synchronization. Lock elision is supported, which can remove the lock and unlock operations for critical sections, which fit in a single transaction. Our multi-threaded approach can switch between automatic and explicit transactions, when it encounters a transaction boundary instruction. Therefore, we can use transactions for synchronization in their full extend.

10.2.4. COTS Fault Tolerance with Intel TSX

Haas et al. [24, 25, 26] suggests the use of Intel TSX to provide fault tolerance on COTS Intel processors. Instrumentation or an additional compiler step is used to split an application into sections, which are executed redundantly in a leading and a trailing process. On the trailing core, these sections are encapsulated in transactions. As both processes perform all writes, separate memory is required for both of them. However, read-only pages can be shared. The approaches in this thesis do not require additional memory, as only one redundant core actually performs the writes. Our HTM ensures input duplication.

A checksum is calculated over all memory writes and register values. The leading process writes these checksums into a shared FIFO queue after each section. The transactions on the trailing process first verify the checksum and commit only if

they match. In case of a checksum mismatch, the current transaction is aborted. Then the leading process is terminated and restarted from the current state of the trailing thread. Recovery by aborting the transaction in the leading process is not possible, as Intel TSX does not support communication of the checksum while the transaction is still running. Therefore, the sections in the leading process are run without transactions. However, our approach can communicate the checksum while the transaction is still running. Therefore, it also uses transactions on the leading cores, as it simplifies recovery.

The FIFO queue can also be used for synchronization, which makes the execution of multi-threaded applications possible. However, this restricts the execution schedule of the trailing process, as the same order of critical sections has to be ensured to guarantee deterministic results. Our multiversioning approach can verify multi-threaded execution even if the order on the trailing cores differs from the leading cores. With Intel TSX, additional efforts are required to use transactions for fault tolerance and synchronization simultaneously. Contrary, multiversioning supports and even favors the use of transactions for synchronization.

10.3. Comparison

In Table 10.1 we compare certain aspects of the presented fault tolerance approaches. It aims to answer the following questions:

- Is the approach implemented in hardware?
An approach, which is implemented in hardware, can cover more faults than a software-based approach. It is also more likely to recover from an error, as the recovery logic cannot be destroyed by a transient fault.
- Does the approach support multi-threaded shared memory applications?
It is an obvious advantage if an approach can execute multi-threaded applications, as otherwise their performance is massively reduced. Some approaches may even require removal of all threading functions, making it complicated to run the application at all.
- Can the approach be switched off to improve performance?
Not all applications might require fault tolerance. Therefore, it is advantageous to disable fault tolerance to improve performance. Some approaches can be disabled, but offer little advantage when doing so, as their hardware cannot be reused.
- Does the approach profit from heterogeneous multi-cores?
Heterogeneous multi-cores offer good performance and low power consumption simultaneously by containing both high-power, fast and low power slow cores. Some approaches can profit from this arrangement by accelerating the slower

Approach	Hardware	Multi-threading	Switchable	Heterogeneous	Recovery	TM for Implementation	TM for Synchronization	Loose Coupling	Memory Consumption
DMR Lockstep	✓	✓?	✓	✗	✗	✗	?	✗	1x
TMR Lockstep	✓	✓?	✗?	✗	✓	✗	✓?	✗	1x
Diva Checker [12]	✓	✗	✗	✓	✓	✗	✗	✗	1x
AR-SMT [53]	✓	✗	✓?	✗	✓	✗	✗	✓	2x
Slipstream [50, 60]	✓	?	✓?	✗	✓	✗	?	✗	2x
ReStore [64]	✓	✗?	✓?	✗	✓	✗	✗	-	1x
SRT [51]	✓	✓?	✓?	✗	✗	✗	✓?	✗	1x
CRT [43]	✓	✓	✓?	✓?	✗	✗	✓?	✗?	1x
SRTR [63]	✓	✓?	✓?	✗	✓	✗	✓?	✗	1x
FaultTM-multi [85]	✓	✓	✓?	✗	✓	✓	✓	✗	1x
PLR [55, 56, 57]	✗	✗	✓	✗	✓	✗	✗	✓	2x-3x
SWIFT [52]	✗	✓?	✓	✗	✗	✗	?	✗	1x
HAFT [39]	✗	✓	✓	✗	✓	✓	✗	✗	1x
COTS Intel TSX [24, 25, 26]	✗	✓	✓	✓?	✓	✓	✗	✓	2x
Ours single-threaded	✓	✗	✓	✓	✓	✓	✓	✓	1x
Ours multi-threaded	✓	✓	✓	✓	✓	✓	✓	✓	1x

Table 10.1.: The existing fault tolerance approaches differ in various aspects and features. Most papers did not consider all aspects. If an aspect is missing in the source and not easily deducible, it is marked with a question mark. In some cases, there is a tendency, i.e. there is nothing, which would make the feature impossible, but it is not clear, how it would be done. These are marked with the tendency and a question mark. ReStore executes fault-free instructions only once. Therefore, coupling does not apply to it, which was marked with a hyphen. Our approaches perform the best in regards to the requirements of the considered embedded system.

core with information from a redundant instance running on the faster core. Therefore, the power consumption is lower than a configuration with just fast cores without losing performance.

- Does the approach feature integrated recovery?
While fault detection can prevent errors in persistent storage, recovery is even more useful, as it enables uninterrupted use of the system. Approaches, which feature integrated recovery, are easier to apply to a system. Other approaches require an additional checkpointing mechanism for recovery, which increases complexity and introduces additional overhead.
- Does the approach use transactional memory for its implementation?
An approach, which reuses parts of a transactional memory, might be easier to implement on a system, which already features an HTM.
- Does the approach support the use of transactional memory for synchronization, while it is active?
Some approaches prevent the use of transactional memory. This limits the possible applications, which can run on such a system.
- Is the approach loosely-coupled?
Loosely-coupled approaches are less restrictive for the implementation of the microarchitecture. They can also offer performance advantages, as the slow-down of one redundant instance does not necessarily affect the other.
- How high is the main memory overhead?
Some approaches require a separate memory region for the redundant instance. This increases production cost, as customers will expect the same amount of usable memory as in non-redundant systems.

For our use case in an embedded system with HTM, we consider the answer “yes” to be optimal. However, different use cases can have different requirements. For example, if the processor cannot be replaced and no hardware-based approach is implemented in the given processor, a software-based approach is required. If no HTM support is available, the implementation of approaches, which use transactional memory, becomes more expensive.

10.4. Summary

There are multiple aspects, which should be considered, when choosing a fault tolerance solution. Our approaches can not only detect errors, but also recover from them without requiring memory for a second instance of the process. Additionally, they are implemented in hardware, which makes the detection and recovery logic itself more resilient to errors than a software approach. Existing HTM logic can be

reused for fault tolerance. Transactional memory can also be used for synchronization. We make use of heterogeneous multi-cores by accelerating the slower cores with information from the leading execution. The loose coupling results in good overall performance even if the slow core cannot be accelerated in a small section. Fault tolerance can also be turned off to increase performance. The simple variant of our approach does not support multi-threaded applications. However, the more complex multiversioning variant does.

Several other fault tolerance approaches exist, but all differ in significant ways from our approaches. Lockstep and TMR are tightly-coupled and cannot profit from heterogeneous multi-cores. The Diva Checker requires a specially designed checker core. AR-SMT and the Slipstream processor require additional memory and cannot profit from heterogeneous multi-cores. ReStore only detects part of all errors, as it is symptom instead of redundancy-based. SRT, CRT and SRTR depend on the SMT capabilities of a processor and cannot be applied to simple embedded systems. FaultM-multi tightly couples transactions on the redundant cores, which restricts scheduling and the use of heterogeneous multi-cores. SWIFT and HAFT duplicate instructions on a single core, which makes them depended on the superscalar capabilities of a processor that are often not available on simple embedded systems. PLR suffer from a long error detection latency, as comparisons only happens at system calls. Software fault tolerance using Intel TSX requires additional memory and a complicated recovery, as Intel TSX does not support checksum transfer in running transactions.

11

Summary and Conclusion

This chapter summarizes the entire thesis. Additionally, an outlook to possible future work is given. Finally, we conclude the thesis based on the evaluation results.

Table of Contents

11.1 Summary	139
11.2 Future Work	141
11.3 Conclusion	143

11.1. Summary

This thesis presents two novel approaches to fault tolerance, which can not only detect but also recover from faults. The first supports only single-threaded applications, while the second also support shared memory multi-threaded applications. Transactional memory can be used for synchronization in the multi-threaded applications. Both heterogeneous and homogeneous multi-cores are supported, in case of the MicroBlaze even without modifications to the core itself. The error detection latency is kept short to enable use in embedded systems. The approaches protect the processor's pipeline from faults. Other parts of the memory hierarchy have to be protected with alternate means like ECC.

Existing HTM is used for isolation and recovery. We automatically split the execution into transactions, which are executed redundantly on two cores. We use loose coupling, which enables the leading core to run ahead of the trailing core. Both copies use the same memory region. However, conflict detection was modified to avoid conflicts between the redundant transactions. Changes are only written back to memory once.

A checksum is formed over all instruction outputs. These checksums are compared, whenever a transaction on the trailing core finishes. If they mismatch, both the leading and the trailing core roll back to the start of that transaction.

On a heterogeneous multi-core, the fast core is used as leading core and the slow core is used as trailing core. By transmitting the memory addresses of cache misses and branch outcomes from the leading to the trailing core, it is possible to accelerate the slower core to keep up with the faster core.

There are several hardware requirements for this approach to work. The leading core's transactional memory implementation has to support two checkpoints, as the oldest checkpoint cannot be cleaned up before the trailing core confirms the corresponding transaction. Especially on heterogeneous multi-cores, the synchronization of transaction lengths can be challenging. Some HTM implementations cannot guarantee that a transaction, which committed on one core, can also commit on another core. However, our approach requires transactions to also commit on the trailing core if they have committed on the leading core.

The single-threaded variant was evaluated by executing microbenchmarks on a heterogeneous multi-core in the gem5 simulator [16], which was extended to support our approach. We examined both performance and power consumption. In the best case, our approach offers up to three times the throughput of a lockstep system consisting of power-efficient cores with the same total power consumption. Alternatively, it consumes up to 35% less power than a lockstep system consisting of fast cores with the same throughput.

The second approach uses multiversioning to execute multi-threaded applications. Multiversioning supports multiple versions for a single cache line. The trailing core always reads the same data as the leading core, since it accesses the same version of the cache line. Therefore, different execution schedules still lead to the exact same result. Multiversioning still supports conflict detection, which enables applications to use transactions for synchronization.

With multiversioning, leading and trailing cores can be dynamically associated. Therefore, different ratios of leading and trailing cores are possible, which also enables homogeneous systems to profit from the acceleration of the trailing cores. The outcomes of the leading and trailing transactions are compared using checksums. If a mismatch is detected, all leading cores are rolled back to error-free and consistent versions. The trailing cores copy the state from the leading core, when the next transaction is started. Although the versions are created independently, our approach is immune to the Domino Effect [38, p. 209] and can always recover.

For our evaluation and to confirm the correct function of the approach, we implemented it on an FPGA. We use closed-source MicroBlaze [73] cores. The multiversioning is implemented in a custom cache. The cores' trace ports are used to replicate the register set for checkpoint creation. Communication between the cores

and the caches, e.g. to start a new transaction, is handled by a memory-mapped interface. This memory-mapped interface also contains various performance counters. Altogether, this allows us to avoid changes to the MicroBlaze cores.

All caches are part of a coherency module, which ensures the cache coherence required to run shared memory multi-threaded applications. We extend each cache line to contain the additional metadata required for multiversioning. This metadata contains the version number and a pointer to the changed data. Conflict detection is also based on the version number.

We propose various optimizations: Conflict detection can be reduced for automatic transactions. A Bloom filter can be used to reduce conflict detection overhead. Small changes can be stored in place of the pointer, which saves a memory access. It is not necessary to fetch cache lines, which are overwritten entirely. Trailing cores should validate transactions from the same leading core to optimize cache usage. It is not necessary to execute writes, which are not read in the same transaction, on the trailing core.

For evaluation, we ported the PARSEC benchmark suite [15] to our platform. The benchmarks are run bare metal, which required us to implement several APIs. Additionally, we had to provide a custom pthreads implementation. We provide two variants: One uses atomic operations for the baseline, while the other uses transactions for multiversioning. We also converted atomic operations in the benchmarks to transactions in the multiversioning variant.

We measured the performance and error detection latency. The mean slowdown of 2.16 compared to the non-redundant baseline is within the expected range for a redundant approach, which executes the application twice. Performance can be enhanced further by optimizing the benchmarks for our platform. The average error detection latency of 9,335 cycles is considered acceptable for most embedded systems. A fault injection analysis demonstrates that our approach could detect and recover from all injected faults.

11.2. Future Work

While we met our goals for this project, there are still starting points to continue working on the presented approaches. One major drawback of the current FPGA implementation is that it requires twice the memory, although this is not inherent to the approach, as currently metadata is always stored in main memory. However, most of the time only a single version of a cache line exists. Therefore, no metadata is required these versions. One possible solution is to prohibit the eviction of cache lines with multiple versions to memory. Thus, it is sufficient to store metadata in the caches. However, this requires a large last level cache, which is not available in our prototype. If one wants to evict versions to memory, it is still possible to reduce the

memory overhead. Currently, data and metadata are interleaved at cache line level. However, it is also possible to interleave them at page level. If all cache lines in a page only consist of one version, the metadata page does not need to be allocated, as the virtual address can point to zero in the page table. By using virtual addresses in the caches, it is not necessary to allocate the page even if another version is created. Only if this version is evicted to main memory, the page needs to be allocated. A hardware mechanism to detect all zero pages can be implemented to free the page once all additional versions are cleaned up.

Another solution might be the use of an alternate hardware implementation of multiversioning altogether. For example, one can store only a single pointer to the next version per cache line. This reduces overhead and permits unlimited versions per cache line, but increases cache miss latency, as pointer chasing is now required.

A more specialized version of multiversioning can store the trailing versions in an alternate way. Trailing cores only create few versions due to the important writes optimization. Additionally, the versions of the trailing cores are never accessed by other cores. Therefore, it is possible to use a versioning approach, which is more similar to classic HTM approaches, for these versions. This reduces the number of versions handled by multiversioning, which decreases its overhead.

The transactional memory implementation in our FPGA prototype is very basic and lacks some features that can be advantageous for embedded systems. The transactions are limited in size by the read- and write-set. This can lead to crashes if a transaction becomes longer than expected by the programmer, e.g. when more loop iterations are required for an algorithm to converge than predicted. Large numbers of conflicts are also not handled well. Currently, the transaction that detects the conflict aborts. In adverse conditions, this can lead to a cycle of transactions aborting themselves repeatedly. To resolve this issue, we use a backoff mechanism written in software. However, the performance cost can be severe if conflicts occur often. In [46], we investigate the use of a transaction management unit, which can also be combined with multiversioning. This transaction management unit makes unbounded transactions possible, which prevents crashes if unexpectedly large transactions occur. It can also prioritize transactions based on various metrics instead of always aborting the detecting transaction. Therefore, performance is improved and cyclic aborts are prevented without the need of a backoff mechanism.

Currently, we have only studied the implementation of fault tolerance with multiversioning. However, we see many more applications for hardware multiversioning. For example, one can use it for thread-level speculation. In thread-level speculation, multiple cores execute different iterations of the same loop simultaneously. However, conflicts arise if these iterations write the same memory location, e.g. because a register is spilled to the stack. Multiversioning can be used to handle these conflicts and to enable all cores to work on their own version.

Databases like PostgreSQL use multiversioning for concurrent accesses to data [61]. Current implementations are software-based, but we think that it is also possible to leverage hardware multiversioning. A hardware implementation can result in improved performance.

Our current FPGA prototype uses MicroBlaze cores for both leading and trailing. However, a heterogeneous multi-core would be preferable. This can be realized using open-source RISC-V cores. For example, we consider the Boom [11] as fast and the Rocket [11] as slow core suitable. However, major adjustments to our cache implementation and evaluation methodology are required. For example, the proprietary trace interface of the MicroBlaze cores, which is used to save the register set, is not available in the RISC-V cores. Our current compiler also does not support the RISC-V architecture.

We have only evaluated injecting bit-flips to the write port of the register set, but our approach should protect the whole pipeline. Therefore, injecting bit-flips to arbitrary LUTs would be interesting. However, this massively increases the number of injections to get statistically sound results. Additionally, the closed-source nature of the MicroBlaze cores is restricting when attempting such injections. The switch to an open-source RISC-V architecture can help with this aspect, too.

11.3. Conclusion

The evaluation of the single-threaded approach shows that a loosely-coupled approach using heterogeneous multi-cores can offer performance and power consumption advantages. Our multiversioning implementation on an FPGA, which can run complex multi-threaded applications like the PARSEC benchmarks, is proof that the concept can be implemented in hardware. We use the multiversioning as transactional memory for the pthreads implementation, which works well in the evaluation. A modification of the closed-source MicroBlaze cores was not required to achieve this. The fault injection analysis demonstrates that the system can actually detect and correct faults. The performance overhead and error detection latency are suitable for most embedded systems. However, future work is still required for a production-ready system, since the current implementation's complexity and area overhead are high. Related use cases for multiversioning are also worth investigating.

List of Figures

1.1.	A sequence of eight instructions is executed redundantly on a fast and a slow core of a heterogeneous multi-core. If tight coupling is used, the fast core has to wait after each instruction for the cores to stay synchronous. Similarly, it has to wait for the slow core to process the cache miss. However, if loose coupling is used, the fast core can run ahead of the slow core. When the cache miss occurs, it can issue a prefetch on the slow core. Therefore, the time the slow core spends waiting for the cache miss is reduced. Finally, both cores finish the sequence before their counterparts on the tightly-coupled system do.	4
1.2.	The same application is executed in both execution schedules: Thread 1 increments x by one, while Thread 2 calculates the bitwise OR of x and one. Both operations are performed atomically without explicit critical sections. In Execution Schedule A, the final value of x is one, while it is two in Execution Schedule B. Both results are valid for the given application. However, a simple comparator-based fault tolerance approach falsely detects an error if the redundant instances happen to execute different schedules.	5
2.1.	This schematic representation of an FPGA is inspired by the Xilinx Virtex family and its depiction in the tool Vivado. The different components are arranged in repeating rows with larger components like BRAMs and DSPs requiring multiple rows for one instance. Switch matrices provide the connectivity between components. CLBs, BRAMs and DSPs are interleaved, while I/O is located in discrete columns. Multiple rows form a clocking region, which shares its global clocks. However, every column contains additional clocking resources, which can be used to select between them. This schematic is reduced massively in scale. An actual FPGA contains more wires per switch matrix, more components per row, more rows per clock region and also multiple clock regions.	16

-
- 3.1. The cores and data caches are extended to support transactional memory and multiversioning. The register snapshot can also be stored external to the core, as its access latency is not performance critical. The sphere of replication, marked by the dotted line, covers the pipeline in the cores. The remaining components are protected by ECC. 22
- 3.2. The single-threaded application is split into transactions TX_i, which are executed on a leading core and a trailing core. For some time the checksums match, but after TX₄ a bit-flip causes a mismatch. This results in a rollback and a restart of TX₄. The delay between the redundant executions is called slack. 24
- 4.1. The automatic transaction boundaries of an exemplary execution are shown above. At first, the CPU switches from supervisor to user mode with fault tolerance enabled. This causes the transaction TX₁ to start. The program begins by executing four memory accesses. After the last store, the cache is full and every cache line is part of either the read- or write-set. Thus, TX₁ commits and TX₂ starts immediately. The cache is still full. However, the cache lines are no longer part of the read- or write-set. Hence, they could be evicted and no additional commit is necessary because of them. In the further execution, the program performs some arithmetic operations. To keep the transaction length roughly equal, the system was configured to limit transactions to six instructions. Therefore, TX₂ commits after the subtraction and TX₃ starts. Finally, the program executes a system call. TX₃ commits before the switch to supervisor mode and no further transaction is started, as the kernel is not protected by this approach. 29
- 4.2. In order to accelerate the trailing core, data is transmitted from the leading core. Perfect prefetching (see Section 4.3.1) uses the memory addresses from the leading core's load store queue to improve the trailing core's prefetcher. Branch outcome forwarding (see Section 4.3.2) uses the branch outcomes from the leading core's commit stage to improve the trailing core's branch predictor. Both solutions require an additional queue between the cores to cover the slack. . . 31

-
- 4.3. The placement of a single-linked list, which was allocated incrementally over the execution of the application, is shown above. Even though, there are only a few discontinuities, a stride prefetcher can only prefetch a fraction of the nodes, as it requires two identical offsets in a row to detect the pattern. Employing a more aggressive prefetching strategy like always fetching the next line, might work in this case. However, it is also possible that the linked list is stored in reverse or completely discontinuous. In this case, the aggressive prefetching strategy would actually hurt performance. Storing additional information about the ordering in the prefetcher is also unlikely to help, as the list was already evicted from cache, so the information in the prefetcher would most likely be evicted, too. However, in our approach, perfect prefetching is possible by forwarding the accessed addresses from the leading to the trailing core. 32
- 5.1. The array load in the source code above can be optimized with a stride prefetcher. The first three loads are issued regularly and cause two cache lines to be fetched. After these loads, the stride prefetcher notices that the address appears to increase by 0x20. Therefore, it predicts which cache lines will be accessed next and prefetches them. After three cache lines, a page boundary is encountered, which prevents further prefetches. 39
- 5.2. Three configurations were evaluated. A lockstep system consisting of two Cortex-A7 and a lockstep system consisting of two Cortex-A15 form the baseline. For our approach a combination consisting of one Cortex-A15 and one Cortex-A7 are evaluated. 43
- 5.3. The microbenchmarks' throughput is shown on the y-axis and the corresponding energy consumption per run on the x-axis. The microbenchmarks were executed on a lockstep system consisting of two Cortex-A7, another lockstep system consisting of two Cortex-A15 and our approach, using a Cortex-A15 as leading core and a Cortex-A7 as trailing core. The clock frequency of the cores was varied in their frequency ranges (Cortex-A7: 500-1300 MHz, Cortex-A15: 700-1900 MHz) in 100 MHz steps for the lockstep systems. For our approach, the trailing core's frequency was fixed at 1300 MHz, while the leading core's frequency was varied from 700 MHz to 1900 MHz. The stride prefetcher is disabled. 44

-
- 5.4. The microbenchmarks' throughput is shown on the y-axis and the corresponding energy consumption per run on the x-axis. The microbenchmarks were executed on a lockstep system consisting of two Cortex-A7, another lockstep system consisting of two Cortex-A15 and our approach, using a Cortex-A15 as leading core and a Cortex-A7 as trailing core. The clock frequency of the cores was varied in their frequency ranges (Cortex-A7: 500-1300 MHz, Cortex-A15: 700-1900 MHz) in 100 MHz steps for the lockstep systems. For our approach, the trailing core's frequency was fixed at 1300 MHz, while the leading core's frequency was varied from 700 MHz to 1900 MHz. The stride prefetcher is enabled. 45
- 6.1. This incorrect execution can occur if the order of the transactions is not preserved between leading and trailing. Thread 1 wants to execute a transaction that increments data by 1. Thread 2 wants to execute a transaction that xors data by 1. If thread 1 is executed first (leading case in this figure), the final result is 0. On the other hand if thread 2 is executed first (trailing case in this figure), the final result is 2. This causes a mismatch in checksums and thus a rollback. . . . 51
- 6.2. The same application as in Figure 6.1 is shown here, but this time executed with multiversioning. From the perspective of the software, the execution on the leading core behaves the same. The result after the first transaction is stored in version 1 and the result after the second transaction in version 2. Even if the thread 2 is executed first in the trailing execution, it still loads version 1 as base. This is possible, as the version is still retained from the leading execution. Thus, the final result is also 0. Version 1 is validated later and the speculative resources are dropped, as version 2 is already available. . . . 51

- 6.3. Multiversioning can store different values for the same address. Bold vertical lines represent the different versions for the memory location 0x100. The labels above them indicate the version and the stored value. Note that the x-axis is not a time axis and all these versions exist at the same point in time (after the commit of the third leading transaction). There is a total of 4 different versions of the memory location 0x100. The version V0 contains the initial value, with which the memory was initialized at program start. The versions V1 and V2 both contain the value 7, which was written by the first transaction. There are separate versions for the leading and trailing transaction. The second transaction only reads the memory location (resulting in the value 7). Therefore, no separate versions were created. The third transaction was only executed by the leading core. It is still waiting for its execution by the trailing core. It adds 8 to the value, which results in another version V4 containing 15. Once the corresponding trailing transaction starts, it will be ordered between the second and third leading transactions. It will also add 8 to the value, resulting in version V3 (not shown). However, as the leading transaction has already committed, its result will not change. Thus, both transactions produce matching results. 53
- 6.4. Each core can be in one of several modes. There can be either an active transaction or not. Outside of transactions, redundant execution and automatic transaction launches are not possible. If there is a transaction, its bounds can be either manual or automatic. Independently, redundancy can be enabled or not. If redundancy is enabled, the core can be either a leading or a trailing core. 54
- 6.5. An example how the version numbering works is shown above. The columns (except for the leftmost labels) represent different versions, while the rows (except for the three top labels) represent different operations. The affected versions regarded by the operation are marked with boxes and circles. Each version is uniquely identified by a combination of its *timestamp*, the *leading* bit and the *core id*. If a leading transaction attempts to read, it considers all versions that were written by any leading transaction. Those are marked by blue boxes. The actual instruction uses the value from the most recent version. It writes to the version (light green circle) indexed by the final timestamp (all bits are 1) and its *core id* (here 1). On commit, the *timestamp* is changed to the current timestamp (here 4568, dark gray box), resulting in the dark green circle. A conflict occurs if an accessed version has the final *timestamp*, is *leading* and the *core id* does not match. The versions, which cause this, are marked by red boxes. Trailing transactions adopt the *timestamp* from the corresponding leading transaction and write directly to this *timestamp*. They read either their version or the most recent previous leading version. . . 57

-
- 6.6. At first, every transaction is confirmed without any errors. However, a bit-flip occurs in TX1.3, which makes global rollback necessary. At this point, every instruction executed in the gray area could possibly be corrupted. It does not matter that the result of the leading and trailing core match for TX1.4, as the bit-flip could have occurred in the leading core, which causes both leading and trailing core to read incorrect data in TX1.4. It is necessary to wait for the trailing cores to confirm TX2.2 and TX3.3, as an error in those transactions would cause a more comprehensive rollback. However, it does not matter that TX3.4 and TX3.5 were not confirmed by the trailing core yet. After the rollback point is determined and all active transactions are aborted, the execution is restarted at TX1.3', TX2.2' and TX3.3'. 65
- 7.1. The memory hierarchy of our system is shown above. MicroBlaze cores are used as processing units. Each core is connected to a block memory, a private instruction cache and a private data cache. All caches are contained in a coherency module. This module is connected to a UART controller and two DDR4 controllers. All components are also connected to various hardware debuggers (not shown). 69
- 7.2. Each cache line contains persistent (will be evicted to RAM) data. The metadata of the cache line itself (e.g. the tag) is not shown. Half of the cache line is used for the data (blue) of the safe version. This data is split into high and low words. The second half is used for versioning information. First, mode & index (red) are stored. Then, 5 version numbers (yellow) follow. Lastly, addresses (green) for all versions except for the safe version are retained. These are also split into high and low words. 71
- 7.3. The **Transaction Control Register** consists of multiple bits with separate functionality. The bits for commit and abort are write-only, as they are only used to invoke momentary actions. The other used bits can be both read and written. The unused bits return zero on read and should be set to zero on write. 77

- 8.1. The program shown here consists of three threads, which protect a critical section with a non-recursive mutex. The threads operate on the same memory, but their current view (gray boxes) can differ, since writes only propagate on commit. The lines of memory locations, which are added to the read- or write-set, are marked in a darker color. Every thread first locks the mutex, then increments data by 1 and finally unlocks the mutex. Initially, all threads appear to successfully lock the mutex, as the mutex is free in version 0. However, when attempting to commit, only one thread can succeed (here thread 1), as they have all read and written the same memory location. Therefore, the other threads roll their transactions back. Now thread 1 has locked the mutex and this state is also globally visible. Thus, it can execute its critical section. The other threads now read version 1, in which the mutex is locked and abort. After completing the critical section thread 1 unlocks the mutex again. The other threads only notice this after the transaction encapsulating the unlock operation is committed. Afterwards, they again compete for the mutex. This time thread 3 wins. 99
- 9.1. These charts show the speedup of the various PARSEC benchmarks in the different configurations at a certain core count. All speedups are relative to the single-threaded execution without redundancy and consider only the region of interest. For the variant without redundancy, the benchmarks were launched with the core count as thread count parameter. For the variant with redundancy, half the core count (i.e. the leading core count) was used as parameter. The lower bound of the expected range is the execution without redundancy repeated twice. The upper bound of the expected range is the execution without redundancy executed twice in parallel with half the core count. Note that especially at low core counts the expected range is hidden behind the line for redundancy for some benchmarks, as the runtimes are so close together. 112
- 9.2. This bar chart shows the speedups of various configurations relative to the single-threaded baseline without transactions or redundancy. The configurations were run on six leading and six trailing cores with multiversioning enabled. Only the region of interest is considered. Each bar represents the median runtime for that benchmark configuration. The error indicators extend to the slowest and fastest runtimes, respectively. The bars are in the same order as the legend entries. . . 117

-
- 9.3. The calculation of the error detection latency is based on instructions and transaction commits. Every leading transaction (blue) is examined together with its corresponding trailing transaction (yellow). The time span between every instruction and the commit of the trailing transaction is calculated. Cycles, in which no instruction was completed, are ignored. The maximum latency is the longest of these time spans, i.e. from the first instruction to the commit of the trailing transaction. The average latency is the average of all these time spans. Note that for the same execution of the same transactions the maximum latency is roughly double the average latency due to the measurement procedure. To aggregate the values for the whole benchmark execution, the weighted average and the maximum are used respectively. 118
- 9.4. Without redundancy, the effect of injected faults can be observed. Each benchmark was executed 50 times and a single bit-flip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. Rollbacks can only occur in the redundant variant with multiversioning. 121
- 9.5. This bar chart shows the results of the fault injection, when redundancy with multiversioning is used. Each benchmark was executed 50 times and a single bit-flip was injected to a random core in each run. After the benchmark completed or timed out, the output was validated and classified as correct, faulty output, crash or freeze. In this evaluation, every output was correct. Sometimes, a global rollback was required to achieve this. 121

List of Tables

5.1. Specifications of the Cortex-A7 and Cortex-A15	38
7.1. The mapping of performance counters into memory is shown above. All addresses are relative to the base address (0xF0000000). Depending on their size, either 4 or 8 bytes are dedicated for one counter, even if this results in some unused bits. This enables easy access with regular integer operations.	86
8.1. Some functions had to be provided or modified for the execution of certain benchmarks. While some functions were implemented fully besides error conditions, other provide only the subset of the functionality, which is required by the benchmarks. Two functions are only stubs, returning zero or empty, and one function would crash with an error message if it was called. For functions, which only required additional synchronization, wrappers are used.	95
9.1. The average error detection latency is the average number of cycles between every instruction and its corresponding checksum comparison. The maximum latency spans from the first cycle in a leading transaction to the checksum comparison of the corresponding trailing transaction. These values were measured for the whole benchmark with 6 leading cores and 6 trailing cores.	119
10.1. The existing fault tolerance approaches differ in various aspects and features. Most papers did not consider all aspects. If an aspect is missing in the source and not easily deducible, it is marked with a question mark. In some cases, there is a tendency, i.e. there is nothing, which would make the feature impossible, but it is not clear, how it would be done. These are marked with the tendency and a question mark. ReStore executes fault-free instructions only once. Therefore, coupling does not apply to it, which was marked with a hyphen. Our approaches perform the best in regards to the requirements of the considered embedded system.	135

List of Source Code Listings

4.1.	This listing contains a recursive implementation of the quicksort algorithm. First a pivot element is selected. The vector is then split into a vector with all elements smaller than the pivot and another vector with all elements greater or equal to the pivot. The algorithm is then applied recursively to both vectors.	32
8.1.	Implementation of atomic fetch and increment using transactions . .	104
8.2.	Implementation of atomic compare and exchange using transactions	105
8.3.	Implementation of the <code>Swap</code> function of the <code>AtomicPtr</code> class in the benchmark <i>canneal</i> using atomics	105
8.4.	Implementation of the <code>Swap</code> function of the <code>AtomicPtr</code> class in the benchmark <i>canneal</i> ported to transactions	106
9.1.	Selected code sample of the <i>streamcluster</i> benchmark	114
9.2.	Selected code sample of the <i>streamcluster</i> benchmark after optimization to reduce the number of barriers	114

Bibliography

- [1] *1:4.4.2-subuntu1 : protoize : i386 : Lucid (10.04) : Ubuntu*. <https://launchpad.net/ubuntu/lucid/i386/protoize/1:4.4.2-subuntu1>. Accessed: 2021-06-18.
- [2] Rico Amslinger, Christian Piatka, Florian Haas, Sebastian Weis, Theo Ungerer, and Sebastian Altmeyer. “Hardware Multiversioning for Fail-Operational Multithreaded Applications”. In: *32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2020, pp. 20–27. DOI: 10.1109/SBAC-PAD49847.2020.00014.
- [3] Rico Amslinger, Christian Piatka, Florian Haas, Sebastian Weis, Theo Ungerer, and Sebastian Altmeyer. “Multiversioning Hardware Transactional Memory for Fail-Operational Multithreaded Applications”. In: *Journal of Parallel and Distributed Computing* (in review 2021). ISSN: 0743-7315.
- [4] Rico Amslinger, Sebastian Weis, Christian Piatka, Florian Haas, and Theo Ungerer. “Redundant Execution on Heterogeneous Multi-cores Utilizing Transactional Memory”. In: *International Conference on Architecture of Computing Systems (ARCS)*. Springer. 2018, pp. 155–167. DOI: 10.1007/978-3-319-77610-1_12.
- [5] ARM Limited. *Cortex-A78AE - Arm@*. <https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a78ae>. Accessed: 2021-06-27.
- [6] Arm Limited. *Arm Cortex-A78AE Core Technical Reference Manual Revision r0p1. 1.1.7 Fault reaction*. <https://developer.arm.com/documentation/101779/0001/Functional-description/Split-Lock-feature/Implementing-Split-Lock/Fault-reaction?lang=en>. Accessed: 2021-06-18.
- [7] Arm Limited. *Arm Cortex-A78AE Core Technical Reference Manual Revision r0p1. 1.1 Implementing Split-Lock*. <https://developer.arm.com/documentation/101779/0001/Functional-description/Split-Lock-feature/Implementing-Split-Lock?lang=en>. Accessed: 2021-06-18.
- [8] Arm Limited. *ARM Synchronization Primitives Development Article. Exclusive monitors*. <https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/exclusive-accesses/exclusive-monitors>. Accessed: 2021-06-18.
- [9] Arm Limited. *Cortex-M33 Dual Core Lockstep. Application Note*. <https://documentation-service.arm.com/static/5ed13ccfca06a95ce53f9129?token=>. Accessed: 2021-06-18.

-
- [10] Arm Limited. *Cortex-R - Arm Developer*. <https://developer.arm.com/ip-products/processors/cortex-r>. Accessed: 2021-06-18.
- [11] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. “The Rocket Chip Generator”. In: *EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-17* (2016).
- [12] Todd M. Austin. “DIVA: A Dynamic Approach to Microprocessor Verification”. In: *Journal of Instruction-Level Parallelism* 2 (2000). ISSN: 1942-9525.
- [13] Todd M. Austin and Gurindar S. Sohi. “Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency”. In: *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO)*. IEEE. 1995, pp. 82–92. DOI: 10.1109/micro.1995.476815.
- [14] Jean-Loup Baer and Tien-Fu Chen. “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. ACM. 1991, pp. 176–186. ISBN: 0897914597. DOI: 10.1145/125826.125932.
- [15] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, 2011.
- [16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. “The gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.
- [17] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692.
- [18] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, Gilles Sassatelli, David Novo, Lionel Torres, and Michel Robert. “Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration”. In: *10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. IEEE. 2016, pp. 201–208. DOI: 10.1109/MCSOC.2016.20.
- [19] Luis Ceze, James Tuck, Călin Caşcaval, and Josep Torrellas. “Bulk Disambiguation of Speculative Threads in Multiprocessors”. In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2006, pp. 227–238. DOI: 10.1109/ISCA.2006.13.
- [20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. Cambridge, 2009. ISBN: 9780262033848.

- [21] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. 3rd ed. London: Oliver and Boyd, 1948. OCLC: 14222135.
- [22] Philip J. Fleming and John J. Wallace. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Communications of the ACM* 29.3 (1986), pp. 218–221. ISSN: 0001-0782. DOI: 10.1145/5666.5673.
- [23] Leo J. Guibas and Robert Sedgewick. “A Dichromatic Framework For Balanced Trees”. In: *19th Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3.
- [24] Florian Haas. “Fault-tolerant Execution of Parallel Applications on x86 Multi-core Processors with Hardware Transactional Memory”. PhD thesis. University of Augsburg, 2019.
- [25] Florian Haas, Sebastian Weis, Stefan Metzloff, and Theo Ungerer. “Exploiting Intel TSX for Fault-Tolerant Execution in Safety-Critical Systems”. In: *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2014, pp. 197–202. DOI: 10.1109/DFT.2014.6962083.
- [26] Florian Haas, Sebastian Weis, Theo Ungerer, Gilles Pokam, and Youfeng Wu. “Fault-Tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support”. In: *International Conference on Architecture of Computing Systems (ARCS)*. Springer. 2017, pp. 16–30.
- [27] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Computing Surveys* 15.4 (1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291.
- [28] Richard W. Hamming. “Error Detecting and Error Correcting Codes”. In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1950.tb00463.x.
- [29] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. 2nd ed. San Rafael: Morgan & Claypool Publishers, 2010. ISBN: 9781608452361. DOI: 10.2200/S00272ED1V01Y201006CAC011.
- [30] Maurice Herlihy and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In: *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*. ACM. 1993, pp. 289–300. DOI: 10.1145/165123.165164.
- [31] Charles A. R. Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (1962), pp. 10–15. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10.
- [32] IEEE, The Open Group. <pthread.h>. <https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html>. Accessed: 2021-06-18. 2018.

-
- [33] IEEE, The Open Group. *pthread_getconcurrency*. https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_getconcurrency.html. Accessed: 2021-06-18. 2018.
- [34] *Implementing Barriers : 15-418 Spring 2013*. <http://15418.courses.cs.cmu.edu/spring2013/article/43>. Course. Carnegie Mellon University. Accessed: 2021-06-18. 2013.
- [35] *Itanium C++ ABI. 5.1 External Names (a.k.a. Mangling)*. <https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling>. Accessed: 2021-06-18.
- [36] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 1st ed. Englewood Cliffs: Prentice Hall, 1978. ISBN: 9780131101630.
- [37] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. “Dynamic Hammock Predication for Non-predicated Instruction Set Architectures”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 1998, pp. 278–285. DOI: 10.1109/PACT.1998.727261.
- [38] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. 1st ed. San Francisco: Morgan Kaufmann, 2007. ISBN: 9780120885251.
- [39] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. “HAFT: Hardware-Assisted Fault Tolerance”. In: *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*. ACM. 2016. DOI: 10.1145/2901318.2901339.
- [40] *ld(1) - Linux manual page*. <https://man7.org/linux/man-pages/man1/ld.1.html>. Accessed: 2021-06-18. 2021.
- [41] ARM Limited. *AMBA® AXI and ACE. Protocol Specification*. <https://developer.arm.com/documentation/ih0022/latest/>. Accessed: 2021-06-18. 2021.
- [42] *llvm-project/cxa_guard_impl.h at main · llvm/llvm-project · GitHub*. https://github.com/llvm/llvm-project/blob/main/libcxxabi/src/cxa_guard_impl.h. Accessed: 2021-06-18. 2020.
- [43] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. “Detailed Design and Evaluation of Redundant Multithreading Alternatives”. In: *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2002, pp. 99–110. DOI: 10.1109/ISCA.2002.1003566.
- [44] Shubu Mukherjee. *Architecture Design for Soft Errors*. 1st ed. Burlington: Morgan Kaufmann, 2008. ISBN: 9780123695291. DOI: 10.1016/B978-0-12-369529-1.X5001-0.
- [45] William Wesley Peterson and Daniel T. Brown. “Cyclic Codes for Error Detection”. In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1961.287814.

- [46] Christian Piatka, Rico Amslinger, Florian Haas, Sebastian Weis, Sebastian Altmeyer, and Theo Ungerer. “Investigating Transactional Memory for High Performance Embedded Systems”. In: *International Conference on Architecture of Computing Systems (ARCS)*. Springer. 2020, pp. 97–108. DOI: 10.1007/978-3-030-52794-5_8.
- [47] *printf(3) - Linux manual page*. <https://man7.org/linux/man-pages/man3/printf.3.html>. Accessed: 2021-06-18. 2021.
- [48] *pthread_yield(3) - Linux manual page*. https://man7.org/linux/man-pages/man3/pthread_yield.3.html. Accessed: 2021-06-18. 2021.
- [49] *pthreads(7) - Linux manual page*. <https://man7.org/linux/man-pages/man7/pthreads.7.html>. Accessed: 2021-06-18. 2021.
- [50] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. “A Study of Slipstream Processors”. In: *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. ACM. 2000, pp. 269–280. DOI: 10.1145/360128.360155.
- [51] Steven K. Reinhardt and Shubhendu S. Mukherjee. “Transient Fault Detection via Simultaneous Multithreading”. In: *Proceedings of 27th International Symposium on Computer Architecture (ISCA)*. ACM. 2000, pp. 25–36. DOI: 10.1145/339647.339652.
- [52] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. “SWIFT: Software Implemented Fault Tolerance”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.
- [53] Eric Rotenberg. “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors”. In: *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE. 1999, pp. 84–91. DOI: 10.1109/FTCS.1999.781037.
- [54] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. “Trace Processors”. In: *Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO)*. IEEE. 1997, pp. 138–148. DOI: 10.1109/MICRO.1997.645805.
- [55] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. “PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures”. In: *IEEE Transactions on Dependable and Secure Computing* 6.2 (2008), pp. 135–148. ISSN: 1545-5971. DOI: 10.1109/TDSC.2008.62.
- [56] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. “Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance”. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2007, pp. 297–306. DOI: 10.1109/DSN.2007.98.

- [57] Alex Shye, Vijay Janapa Reddi, Tipp Moseley, and Daniel A. Connors. “Transient Fault Tolerance via Dynamic Process-Level Redundancy”. In: *Proceedings of Workshop on Binary Instrumentation and Applications (WBIA)*. 2006.
- [58] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. 1st ed. Vol. 4. 1. San Rafael: Morgan & Claypool Publishers, 2009. ISBN: 9781598299540. DOI: 10.2200/S00192ED1V01Y200904CAC005.
- [59] Stratus Technologies, Inc. *Stratus® ftServer® 6500 Hardware*. <https://www.stratus.com/assets/6500hw.pdf>. Accessed: 2021-06-18. 2002.
- [60] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. “Slipstream Processors: Improving both Performance and Fault Tolerance”. In: *ACM SIGPLAN Notices* 35.11 (2000), pp. 257–268. ISSN: 0362-1340. DOI: 10.1145/356989.357013.
- [61] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 12: 13.1. Introduction*. <https://www.postgresql.org/docs/12/mvcc-intro.html>. Accessed: 2021-06-18.
- [62] *Update from 2018.3 to 2019.1 breaks DDR4 MIG*. <https://forums.xilinx.com/t5/Memory-Interfaces-and-NoC/Update-from-2018-3-to-2019-1-breaks-DDR4-MIG/m-p/982425>. Accessed: 2021-06-18. 2019.
- [63] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. “Transient-Fault Recovery Using Simultaneous Multithreading”. In: *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2002, pp. 87–98. DOI: 10.1109/ISCA.2002.1003565.
- [64] Nicholas J. Wang and Sanjay J. Patel. “ReStore: Symptom-Based Soft Error Detection in Microprocessors”. In: *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006), pp. 188–201. ISSN: 1545-5971. DOI: 10.1109/TDSC.2006.40.
- [65] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. “Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations”. In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2013, pp. 215–224. DOI: 10.1109/ISPASS.2013.6557172.
- [66] Robert Wegrzyn, Steve Gross, Vikram Patel, Atilla Bulmus, and Cuauhtemoc Medina Rimoldi. *Safety design for modern vehicles - including EV/HEV*. https://www.infineon.com/dgdl/Infineon-Safety%20Design%20for%20Modern%20Vehicles%20Whitepaper-Whitepaper-v01_00-EN.pdf?fileId=5546d4626cb27db2016d24bcb8b26396&da=t. Accessed: 2021-06-18.
- [67] John William Joseph Williams. “Algorithm 232: Heapsort”. In: *Communications of the ACM* 7 (1964), pp. 347–348. ISSN: 0001-0782. DOI: 10.1145/512274.512284.
- [68] Xilinx, Inc. *AXI Interconnect v2.1. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf. Accessed: 2021-06-18. 2017.

- [69] Xilinx, Inc. *AXI UART Lite v2.0. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf. Accessed: 2021-06-18. 2017.
- [70] Xilinx, Inc. *Integrated Logic Analyzer v6.2. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf. Accessed: 2021-06-18. 2016.
- [71] Xilinx, Inc. *Local Memory Bus (LMB) v3.0. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/lmb_v10/v3_0/pg113-lmb-v10.pdf. Accessed: 2021-06-18. 2016.
- [72] Xilinx, Inc. *MicroBlaze Debug Module v3.2. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf. Accessed: 2021-06-18. 2021.
- [73] Xilinx, Inc. *MicroBlaze Processor Reference Guide*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug984-vivado-microblaze-ref.pdf. Accessed: 2021-06-18. 2019.
- [74] Xilinx, Inc. *MicroBlaze Triple Modular Redundancy (TMR) Subsystem v1.0. Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/tmr/v1_0/pg268-tmr.pdf. Accessed: 2021-06-18. 2019.
- [75] Xilinx, Inc. *UltraScale Architecture Clocking Resources. User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf. Accessed: 2021-06-18. 2020.
- [76] Xilinx, Inc. *UltraScale Architecture Configurable Logic Block. User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf. Accessed: 2021-06-18. 2017.
- [77] Xilinx, Inc. *UltraScale Architecture DSP Slice. User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf. Accessed: 2021-06-18. 2020.
- [78] Xilinx, Inc. *UltraScale Architecture Memory Resources. User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf. Accessed: 2021-06-18. 2021.
- [79] Xilinx, Inc. *UltraScale Architecture SelectIO Resources. User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug571-ultrascale-selectio.pdf. Accessed: 2021-06-18. 2019.
- [80] Xilinx, Inc. *UltraScale Architecture-Based FPGAs Memory IP v1.4. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf. Accessed: 2021-06-18. 2021.
- [81] Xilinx, Inc. *Virtual Input/Output v3.0. LogiCORE IP Product Guide*. https://www.xilinx.com/support/documentation/ip_documentation/vio/v3_0/pg159-vio.pdf. Accessed: 2021-06-18. 2018.

-
- [82] Xilinx, Inc. *xsdserver start*. https://www.xilinx.com/html_docs/xilinx2019_1/SDK_Doc/xsct/miscellaneous/reference_miscellaneous_xsdserver_start.html. Accessed: 2021-06-18. 2019.
- [83] Gulay Yalcin, Osman Unsal, and Adrian Cristal. “FaultTM: Error Detection and Recovery Using Hardware Transactional Memory”. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. EDAA. 2013, pp. 220–225. DOI: 10.7873/DATE.2013.058.
- [84] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, and Mateo Valero. “FaultTM: Fault-Tolerance Using Hardware Transactional Memory”. In: *Pespm 2010 - Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*. 2010.
- [85] Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. “Fault Tolerance for Multi-Threaded Applications by Leveraging Hardware Transactional Memory”. In: *Proceedings of the ACM International Conference on Computing Frontiers (CF)*. ACM. 2013. DOI: 10.1145/2482767.2482773.

A

Bare Metal API

The Xilinx SDK already provides some basic functionality. For example, heap management and a simple UART driver are already included. However, this is not sufficient to run the PARSEC benchmarks. Therefore, we implemented additional APIs and extended existing ones.

Table of Contents

A.1 File System	xxxvii
A.2 Memory Allocation	xxxix
A.3 Compiler Helper	xl
A.4 Various	xl

A.1. File System

Our platform provides a memory file system. Every file consists of a `control_block` structure and its content. The `control_block` stores the following attributes:

next: The blocks are arranged in a single linked list, which is traversed with this pointer.

name: The file name includes the full path of the corresponding file.

start, end: These pointers reference the start and end of the allocated memory region for the file contents. If the end pointer is passed, a new larger memory region needs to be allocated.

size: This is the size of the file content, which can be smaller than the allocated size.

location: This is the current location, at which the file is read or written.

flags: The flags of the last `open` call are preserved.

crc: A CRC checksum enables output validation without downloading files to a host machine.

There is no support for directories at a data structure level. However, some functions operate on directories. These functions assume a directory exists if there is a file name with the directory as prefix. Additional logic is used to translate relative paths.

All files, which are supposed to be included in the binary, are referenced in a macro in an assembler file. This macro creates the necessary description block and copies the file content into the binary. An additional macro is available to create empty files.

The `open` function opens a file for reading or writing. It is implemented by searching all descriptors for the given file name. It is not possible to create new files. We do not allocate additional descriptors for open files. Instead, the current location is stored in the `control_block` corresponding to the file. However, this entails the limitation that a single file cannot be accessed concurrently with multiple descriptors. The `location` and `crc` attributes are reset and the `flags` attribute is set according to the corresponding parameter. Finally, the index, offset by 1000 is returned. This offset is necessary to avoid overlaps with predefined IDs like the standard output stream.

The `close` function closes a previously opened file. It calculates the CRC checksum and outputs a message containing the file name, a pointer to the data, file size and checksum to the UART interface.

The `read` function reads data from a file. It is implemented with a memory copy and increment of the `location` field.

The `write` function writes data to a file. If the allocated area is too short, it is enlarged in powers of two. The data is then copied and the `location` field incremented.

The `stat` and `fstat` functions return information about a file. They first check if a file with the exact name or prefix exists. If the name matches exactly, the flags are set to `file`. If only a prefix matches, they are set to `directory`. Otherwise, they are set to `nonexistent`. The only other real attribute is the file size. All other returned attributes are set to placeholder values.

The `lseek` function changes the current location in a file.

The `access` function checks whether the user has permissions to perform an action on a file. Permissions are not implemented. Therefore, the function always returns zero. However, this implementation is incomplete, as it would also need to check for the existence of parent directories. Additionally, it is not actually possible to write nonexistent files, as the `open` function cannot open them.

The functions `opendir`, `readdir` and `closedir` are used to iterate the content of a directory. They are implemented by iterating over all `control_blocks` and returning the files with matching prefixes. Subdirectories are not handled correctly, but this does not matter for the PARSEC benchmarks.

The `mmap` function is used to map a file into memory. Its implementation checks the allocated data area for sufficient size and enlarges it if necessary. Then the pointer to this area is returned. Many more uses of the `mmap` function, like inter-process communication or ring buffers, exist. However, these are all not possible on our platform, as the MMU is disabled.

The `getcwd` function returns the current working directory. In our implementation, it always returns an empty string, as it is not possible to change the working directory.

The `ftruncate` function sets the size to the current location. As this can also mean that the file is enlarged, the size of the allocated area is checked first and extended as needed. Then the `size` attribute of the `control_block` is set to the current location.

On UNIX based systems, a combination of the functions `link` and `unlink` can be used to rename files. The benchmark *x264* has a case, in which it renames a file. However, this case is not reached during execution. Therefore, we have implemented the function `link` as a stub. Curiously, the function `unlink` is already provided by the standard library shipped with the compiler. We assume that you have to replace this function as well to make file renaming work.

A.2. Memory Allocation

The standard library provided by the Xilinx SDK already provides support for dynamic memory allocation. It is implemented by using a static memory region as heap. We have sized this region sufficiently for the *simmedium* input set of the PARSEC benchmarks.

The standard library was compiled without multi-threading enabled, which resulted in omission of the necessary synchronization. However, we use it on a multi-threaded platform, which requires synchronization for memory management functions. Therefore, the functions `malloc`, `calloc`, `realloc` and `free` have to be encapsulated in a critical section, as they are not thread safe.

The necessary locks can be integrated by using the symbols `__malloc_lock` and `__malloc_unlock`. All memory allocation functions branch to these symbols when entering and leaving the critical section. Contrary to their names, this also applies to `free`. Using these symbols has the advantage that library functions, which have

inlined the memory allocation, are also affected. One example for such a function is `strdup`, which requires memory allocation, as it duplicates the passed string.

Additionally, we decided to align all memory allocations to whole cache lines. This helps to reduce false sharing and leads to more consistent runtimes.

A.3. Compiler Helper

The compiler uses the functions `__cxa_guard_acquire`, `__cxa_guard_abort` and `__cxa_guard_release` to protect the allocation of local static variables [42]. We use a single mutex for all variables to ensure thread safety. This does not affect performance, as there are usually only a few local static variables and they only have to be initialized once. The guard flag is set as expected by the compiler to ensure that the auto-generated code at the beginning of the function, which checks whether the static variables are initialized, works as expected.

A.4. Various

The functions `printf` and `vprintf` have to be encapsulated in a critical section. They make use of a global buffer and are therefore not thread safe. However, according to the documentation [47] they should be thread safe and the PARSEC benchmarks assume that they are thread safe. For our `printf` implementation, we call `vprintf`, as this simplifies the handling of the variable arguments. Note that I/O is not described in detail here, as the functionality provided by the Xilinx SDK is already sufficient.

The function `gettimeofday` is not available by default, as the MicroBlaze processors do not include a real time clock. Our platform does not include a real time clock either. As the PARSEC benchmarks mostly do not care about the exact date but only the difference between timestamps, we provide an implementation based on the *Cycle Count* performance counter (see Section 7.2.7). The benchmark *vips* is an exception, as it writes the current date to the output file. However, we ignore this field when validating the file.

B

Necessary Changes to the Parsec Benchmarks

While porting the Parsec Benchmark Suite to our platform, some changes to the benchmarks were necessary. Many of these changes are required to resolve compiler errors. However, some also correct race conditions, which would intervene with fault injection analysis.

Table of Contents

B.1 blackscholes	xlii
B.2 bodytrack	xlii
B.3 canneal	xliii
B.4 dedup	xliii
B.5 facesim	xliv
B.6 ferret	xlvi
B.7 fluidanimate	xlvi
B.8 freqmine	xlvi
B.9 raytrace	xlvi
B.10streamcluster	xlvi
B.11swaptions	xlvi
B.12vips	xlvi
B.13x264	xlvi

B.1. blackscholes

We corrected the calculations of the start and end index for each thread. The old formulas are:

```
int start = tid * (numOptions / nThreads);  
int end = start + (numOptions / nThreads);
```

However, these result in incorrect results if `numOptions` is not a whole multiple of `nThreads`. Therefore, we changed the order of operations and applied the distributive property:

```
int start = tid * numOptions / nThreads;  
int end = (tid + 1) * numOptions / nThreads;
```

Note that `numOptions` is at most 10,000,000. Therefore, up to 214 threads can be managed with 32 bit integers.

B.2. bodytrack

The benchmark *bodytrack* provides implementations of atomic operations in assembler code for various architectures. However, they are not used, so we removed them. Instead, we provided implementations both with transactions and load linked/store conditional for the compiler-based atomic operations.

The reference counter in the class `FlexImageStore` is accessed concurrently, but not atomically. As this can lead to corrupted outputs, we added the missing atomic operations.

In the `AsyncImageLoader` class, condition variables are used incorrectly. A separate mutex is locked when waiting for the condition variable. This can lead to a race condition on the variables used as condition. We replaced the false mutexes and ensured that the critical section covers the whole condition.

We added “`this->`” in front of the call of the function `AddGaussianNoise` in the generic class `ParticleFilterPthread<T>`. This is necessary to compile the code, since the function is dependent, as the type of the variable `mModel` is the generic parameter.

We removed all `throw` specifiers from destructors, as a subclass is not allowed to throw an exception in its destructor if the destructor of the base class does not. However, this happens in several instances in this benchmark. Alternatively, we added the `noexcept` specifier for the destructor of some classes, which contain members that throw in their destructor.

Instead of casting a reference and then taking the pointer, we take the pointer first and then cast. This avoids an exception if the given reference is not actually an instance of the target class, which can occur regularly in this benchmark.

B.3. *canneal*

The benchmark *canneal* provides implementations of atomic operations in assembler code for various architectures. However, MicroBlaze assembler was not provided. Therefore, we implemented them both with transactions and load linked/store conditional. We removed some unused functions in the `AtomicPtr` class, as these use atomic operations, which we did not implement. Implementing them would not have been challenging, but the correctness of untested atomic operations is always dubious.

We added “`return nullptr;`” to the function `entry_pt`, as it has a return type of `void*` but no return value. Most compilers return the equivalent value of zero in this case. However, the MicroBlaze gcc continues execution with whatever instructions follow the function if it encounters a function with return type but no `return` instruction.

B.4. *dedup*

K & R C refers to an old variant of the C programming language, which was described in the book “The C Programming Language” [36] (first edition only). This variant differs from modern C in significant ways. For example, the type of function parameters is written after the closing parenthesis instead of before the parameter name, which results in functions like the example below.

```
uLong ZEXPORT compressBound (sourceLen)
uLong sourceLen;
{
    return sourceLen + (sourceLen >> 12) + (sourceLen >> 14) + 11;
}
```

This dialect is no longer supported by modern compilers like the MicroBlaze compiler. However, the PARSEC benchmarks still make use of this syntax. There is a tool called *protoize*, which can convert old source code to the new format. However, this tool is no longer available for current operating systems. We were able to find an old version [1] for Ubuntu 10.04, which still runs. This version does not support directories, which makes it necessary to move all files into a single directory and to redistribute them to their respective subdirectories after conversion. We have removed unused K & R C functions altogether.

We replaced `u_long` with `uint64_t`, as `u_long` is actually just 32 bits wide on the MicroBlaze. However, the benchmark uses the size of the variable `len` to read from a file and the intended read size is 8 bytes.

We removed conflicting integer type definitions. These conflicting definitions are probably also the reason why the benchmark has ever worked on 32-bit platforms, as they declare `u_long` as 64 bits wide. However, the MicroBlaze gcc does not allow conflicting definitions.

We have added missing `extern` and `static` keywords in front of some declarations to ensure that they refer to the correct instance.

We initialize `optind` with one instead of zero as it is intended. The more thorough implementations of `getopt` can handle the wrong initialization, but the MicroBlaze implementation cannot.

The *ZLib* library used by the *dedup* benchmarks uses `this` as variable name. However, `this` is a reserved keyword. Therefore, we replaced all usages with a different name.

The *openssl* library is only used for the SHA1 hash function. We removed all unrelated files to avoid complications compiling the unnecessary source code.

We defined the constant `SSIZE_MAX`, since it is missing from the `limits.h` file, which is supposed to define it.

We disabled spin locks and use regular mutexes instead. Spin locks are prone to deadlocks on our platform, as it does not feature timer interrupts.

B.5. facesim

We have renamed the `sync` struct to `facesimSync`, as the name conflicts with a library function `sync`, which prevents the benchmark from compiling.

We do not support the creation of directories on our platform, but the unmodified benchmark handles this case gracefully by assuming that the directory already exists.

The benchmark uses deeply nested includes with relative paths. In some cases, this leads to a path that exceeds the maximum allowed length. We shortened some directory names to avoid this.

We have renamed all header files with file extension “.c” to “.h” to prevent their compilation as source files.

We replaced the file existence check based on `std::ifstream` and `std::ofstream` with one based on the `open` function, since our compiler does not allow the comparison between the streams and zero.

B.6. ferret

We had to pull a union out of a sizeof expression, as type declarations are not allowed in sizeof expressions.

We reordered struct initialization to be in the same order as the field declarations are. Otherwise, a compiler error occurs.

We removed all includes of the file `values.h`, as it does not exist. We include the file `float.h` instead, as it contains the constant `DBL_MAX`, which we assume is defined in `values.h` on the machine of the benchmark author.

Some files are missing required includes. In pure C compilation works, as every symbol is identical to the function name. However, a C++ compiler uses name mangling [35], which prevents the linker from matching the correct function. We inserted “`extern "C"`” in front of the definition and declaration of the affected functions to prevent mangling of their names.

To avoid complications, we compile only the files from the *GNU Scientific Library*, which contain functions that are used by the benchmark *ferret*.

We replaced all thread local variables with pthreads specifics, since the compiler cannot compile thread local variables properly.

We cast pointers of type `void**` to `char**`, when pointer arithmetic is performed, as the compiler does not allow the operations with `void**` pointers.

We replaced the incorrect type `cass_map_t*` with the correct type `bitmap_t*`, since the compiler, in contrast to old C compilers, does not allow implicit casting of these types.

We replaced all calls to the function `lstat` with `stat`, as there are no links in our memory file system.

We removed the file `local.c`, as its functions are never called, but it calls undefined functions.

B.7. fluidanimate

We replaced all calls to the function `posix_memalign` with calls to `malloc`, as on our platform `posix_memalign` is undefined and `malloc` returns cache line aligned memory.

Our platform supports barriers natively. Therefore, we do not use the barrier implementation provided with the PARSEC benchmark suite.

B.8. freqmine

The *freqmine* benchmark does not provide a pthreads implementation. As there is no working OpenMP support in the MicroBlaze compiler, it would be necessary to port the whole benchmark to pthreads. This was not done for this work.

B.9. raytrace

We compiled this benchmark with the `-fno-strict-aliasing` flag, as the function `_mm_and_ps` breaks strict aliasing rules.

We added an explicit check for NaN in the function `_mm_cvtps_epi32`, as the benchmark expects NaN to be a certain integer value (0x80000000).

We clear the `MemoryFrameBuffer` to zero on allocation, as parts of the buffer are never written, since the height is no multiple of the block size. Otherwise, the benchmark's output contains random bytes, which often fail validation.

When the `doneWithFrame` function of `MemoryFrameBuffer` is called, we output its content to a file, as it is discarded otherwise.

We removed several files containing unused functions, as they did not compile. For example, they referenced *OpenGL*, which is not available on the MicroBlaze. The function prototypes and calls in impossible ifs were removed as well.

B.10. streamcluster

We moved the final assignment of `gl_cost_of_opening_x` after the barrier, as other threads read it before the barrier causing a race condition.

We moved the check whether the number of points is less than or equal to `kmax` up, as otherwise already freed memory for the variable `hizs` could get accessed.

We moved static variables in functions out of the function. This prevents possible issues when automatic transactions are not validated (see Section 7.4.1).

Our platform supports barriers natively. Therefore, we do not use the barrier implementation provided with the PARSEC benchmark suite.

We removed all uses of the initialization macros `PTHREAD_MUTEX_INITIALIZER` and `PTHREAD_COND_INITIALIZER`, since instances of our classes are always initialized, as they use explicit default constructors.

B.11. *swaptions*

No changes were necessary to compile and run the benchmark *swaptions*.

B.12. *vips*

As in the benchmark *dedup*, we also converted all K & R C code.

We have renamed variables with names, which are a keyword in the MicroBlaze gcc. This includes `class`, `namespace`, `new`, `or` and `this`.

If a function with return type is missing the return statement, compilers usually behave as if there was a return with an undefined value at the end. However, the MicroBlaze compiler omits the return completely. Therefore, execution continues with the next function in the assembly. We have added return statements to all functions, which were missing them, to restore the expected behavior.

As in the benchmark *ferret*, we also inserted “`extern "C"`” to resolve faulty includes.

We removed some files containing only unused functions, as they cause issues when compiling. However, even with these files removed, there are too many files and the linker arguments exceeds the maximum allowed length. The original build configuration incrementally links the files to avoid this issue. However, this is not easily possible in our build environment. As an alternative, we shortened some directory names to get below the allowed length.

We removed all functionality related to the function `im_system`, as it is not needed and our bare metal implementation does not support processes or pipes.

The function `im_guess_libdir` determines the installation directory by examining environment variables. As this is not possible on our platform, we have modified the function so that it always returns the correct value, which is the empty string.

We removed the `restrict` type qualifier everywhere, as apparently our compiler does not support it.

Sometimes the `#ifdef` blocks for the opening and closing curly bracket are nested differently. With our specific configuration, some isolated brackets occur and cause compiler errors. We corrected the nesting of the `#ifdef` blocks.

The declarations of function pointers are missing the parameter list. Our compiler outputs an error if it encounters a call to such a function pointer with parameters. Therefore, we added the missing parameters to all function pointers.

We removed various sections of code from the library *glib*, as they do not compile. The sections are never executed and only present, since the benchmark’s authors

included the whole library. We also removed the initialization for features, which are otherwise unused.

We removed some guards from the library to enable compilation together with the benchmark in a single pass.

We removed the file `config.h` and all includes of it, as this file is supposed to be generated by the build environment. This is not the case for us, but the file is also not needed for any feature that is actually used.

The files `galias.h` and `galiasdef.c` undefine and redefine symbols to optimize the Procedure Linkage Table. However, our compiler considers this as conflicting definitions, which is not allowed. Therefore, we removed all includes to these files.

We added zero initialization to `GBSearchArray`, when a new array is allocated or the existing one grows.

Even if the `ENABLE-NLS` constant is undefined, some translation code remains. We removed calls to `bindtextdomain` and `bind_textdomain_codeset` to resolve compiler errors.

We removed all plugin logic, since it is not necessary for benchmark execution, but can cause issues.

We removed all calls to the functions `munmap` and `getpagesize`, as the prototype does not use virtual memory. As stated in Section A.1, the function `mmap` is supported. However, it does not create a new virtual memory mapping, but returns a pointer to the file data instead, which should never be freed.

We removed the function `print_links`, since the compiler has trouble with the constant concatenation.

We modified the function `im_concurrency_get` to return the thread count passed to the benchmark. The original implementation reads environment variables and uses the function `sysconf` to get the actual core count, which is not supported on our platform. We replaced all other calls to the function `sysconf` with hard-coded values.

B.13. x264

We clear buffers initialized with the `x264_malloc` function to zero, as the benchmark behaves differently depending on the old values in them. Most likely, there is an uninitialized read somewhere.

We replaced the `endian_fix` and `endian_fix32` functions with the identity function, as no endian fixing is needed. In fact, the values are wrong if their endianness is swapped.

We split the loop in `x264_cqm_init` to prevent it from going out of bounds. We also corrected the deallocation in `x264_cqm_delete`.

We have renamed all header files with file extension “.c” to “.h” to prevent their compilation as source files.

We removed some files, which were not needed and did not compile. They are related to render debug output to the *X window system*.

We renamed the local variable `or`, as `or` is a reserved keyword.

We have added explicit casts, wherever negative constants are used as unsigned values. Without these casts, the compiler complains that the conversion underflows. This is true, but expected, as it is often used to fill the entire binary representation with ones by casting `-1` to an unsigned value.

We corrected the loop bound in the function `x264_encoder_frame_end` to prevent out of bound accesses.